made with doc-o-matic

RAD Studio

Copyright(C) 2009 Embarcadero Technologies, Inc. All Rights Reserved.

Table of Contents

Concepts 1

Debugging C++ Applications with CodeGuard Error Reporting 3

CodeGuard Errors 3

Access Errors 4

Exception Errors 5

Function Failure Errors 6

Resource Errors 7

CodeGuard Overview 9

CodeGuard Warnings 10

Memory Block Comparison Warnings 11

Pathname Merging and Splitting Warnings 11

String Comparison Warnings 11

Developing Database Applications for the Win32 Platform 13

dbGo Overview 14

BDE Overview 15

dbExpress Components 16

Getting Started with InterBase Express 17

dbExpress 4 Feature Overview 22

Blackfish SQL Overview 24

dbExpress Framework 26

dbExpress Framework Compatibility 27

Developing Interoperable Applications 29

Developing COM Applications 29

Developing Reports for Your Win32 Applications 35

Using Rave Reports in RAD Studio 35

Developing Applications with VCL Components 36

VCL Overview 36

Using TEncoding for Unicode Files 38

Components Available Only on Specific OS 39

Developing Web Applications with WebSnap 41

Win32 Web Applications Overview 41

Developing Web Services with Win32 Applications 44

Web Services Overview 44

Developing Windows Applications **45**
Windows Overview 45

Procedures 47

CodeGuard Procedures **48**
Using CodeGuard 48

Database Procedures **50**
Accessing Schema Information 51
Using Data Explorer to Obtain Connection Information 52
Configuring TSQL Connection 53
Connecting to the Application Server using DataSnap Components 55
Debugging dbExpress Applications using TSQLMonitor 55
Executing the Commands using TSQLDataSet 56
Fetching the Data using TSQLDataSet 57
Specifying the Data to Display using TSQLDataSet 58
Specifying the Provider using TLocalConnection or TConnectionBroker 59
Using BDE 60
Using DataSnap 60
Using TBatchMove (BDE) 61
Connecting to Databases with TDatabase 62
Using TQuery (Procedure) 63
Using TSQLQuery 65
Using TSQLStoredProc (Procedure) 66
Using TSQLTable 66
Managing Database Sessions Using TSession 67
Using TSimpleDataSet 68
Using TSimpleObjectBroker 69
Using TStoredProc 69
Using TTable (Procedure) 70
Using TUpdateSQL to Update a Dataset 72
Using dbExpress 72
Adding a New Connection to the Data Explorer 73
Browsing a Database in the Data Explorer 74
Executing SQL in the Data Explorer 74
Modifying Connections in the Data Explorer 75
Connecting to a Database using the dbExpress Driver Framework 76

Interoperable Applications Procedures **79**

Using COM Wizards 79

Reporting Procedures 81

Adding Rave Reports to RAD Studio 81

VCL Procedures 82

Building a Windows "Hello World" Console Application 88
Developing a Windows Application 89
Building Application Menus 90
Building a VCL Forms Application with Decision Support Components 91
Building VCL Forms Applications With Graphics 93
Building a VCL Forms MDI Application Using a Wizard 93
Building a VCL Forms MDI Application Without Using a Wizard 94
Building a VCL Forms SDI Application 96
Creating a New VCL Component 97
Building a VCL Forms ADO Database Application 98
Building a VCL Forms Application 99
Creating Actions in a VCL Forms Application 100
Building a VCL Forms "Hello World" Application 101
Using ActionManager to Create Actions in a VCL Forms Application 102
Building a VCL Forms dbExpress Database Application 103
Building an Application with XML Components 104
Copying Data From One Stream To Another 106
Copying a Complete String List (VCL) 107
Creating Strings 109
Creating a VCL Form Instance Using a Local Variable 110
Deleting Strings 112
Displaying an Auto-Created VCL Form 113
Displaying a Bitmap Image in a VCL Forms Application 114
Displaying a Full View Bitmap Image in a VCL Forms Application 116
Drawing a Polygon in a VCL Forms Application 117
Drawing Rectangles and Ellipses in a VCL Forms Application 118
Drawing a Rounded Rectangle in a VCL Forms Application 118
Drawing Straight Lines In a VCL Forms Application 119
Dynamically Creating a VCL Modal Form 120
Dynamically Creating a VCL Modeless Form 121
Iterating Through Strings in a List 123
Building a Multithreaded Application 124
Writing Cleanup Code 125
Avoiding Simultaneous Thread Access to the Same Memory 125

Defining the Thread Object 126
Handling Exceptions 129
Initializing a Thread 130
Using the Main VCL Thread 131
Waiting for Threads 132
Writing the Thread Function (Procedure) 134
Placing A Bitmap Image in a Control in a VCL Forms Application 135
Reading a String and Writing It To a File 136
Renaming Files 137
Adding and Sorting Strings 138
Creating a VCL Forms ActiveX Button 139
Creating a VCL Forms ActiveX Active Form 140
Building a VCL Forms Web Browser Application 142
Creating an Application that Uses Ribbon Controls 143
Adding Commands to the Ribbon 144

WebSnap Procedures **147**
Building a WebSnap Application 148
Building a WebSnap "Hello World" Application 149
Debugging a WebSnap Application using the Web Application Debugger 150
Using the HTML Tag Editor 151

Web Services Procedure **153**
Building a "Hello World" Web Services Application 153

Reference 155

C++ Reference **156**
Command Line Utilities 156
BCC32, the C++ Command-Line Compiler 159
BRC32, the Resource Shell 163
BRCC32.EXE, the Resource Compiler 165
COFF2OMF.EXE, the Import Library Conversion Tool 166
CPP32.EXE, the C Compiler Preprocessor 167
DCC32.EXE, the Delphi Command Line Compiler 169
GREP.EXE, the text search utility 170
ILINK32.EXE, the Incremental Linker 174
IMPDEF.EXE, the Module Definition Manager 179
IMPLIB.EXE, the Import Library Tool 181
Using Include Files 182
MAKE 183

MAKE Directives 186
MAKE Macros 191
MAKE Rules (Explicit and Implicit) and Commands 193
Message Options 197
Module Definition Files 199
Using Precompiled Header Files 203
RLINK32.DLL, the Resource Linker (C++) 204
TDUMP.EXE, the File Dumping Utility 204
TLIB.EXE, the Library Manager 208
Using TOUCH.EXE 212
TRIGRAPH 213
RC.EXE, the Microsoft SDK Resource Compiler 213
WSDLIMP.EXE, the Command Line WSDL Import Tool 214
C++ Compiler Errors And Warnings (C++) 216
E2066: Invalid MOM inheritance (C++) 256
E2525: You must define _PCH_STATIC_CONST before including xstring to use this feature (C++) 257
E2526: Property 'name' uses another property as getter/setter; Not allowed (C++) 257
E2008: Published property access functions must use __fastcall calling convention (C++) 257
E2122: Function call terminated by unhandled exception 'value' at address 'addr' (C++) 257
E2506: Explicit specialization of 'specifier' is ambiguous: must specify template arguments (C++) 257
E2483: Array dimension 'specifier' could not be determined (C++) 258
E2509: Value out of range (C++) 258
E2510: Operand size mismatch (C++) 258
E2050: __declspec(delphireturn) class 'class' must have exactly one data member (C++) 258
E2530: Unrecognized option, or no help available (C++) 258
E2527: Option 'name' cannot be set via 'name' (C++) 258
E2528: Option 'name' must be set before compilation begins (C++) 258
E2074: Value after -g or -j should be between 0 and 255 inclusive (C++) 259
E2492: Properties may only be assigned using a simple statement, e.g. \"prop = value;\" (C++) 259
E2505: Explicit instantiation requires an elaborated type specifier (i.e.,"class foo<int>") (C++) 259
E2100: Invalid template declarator list (C++) 259
E2102: Cannot use template 'template' without specifying specialization parameters (C++) 260
E2107: Invalid use of template 'template' (C++) 260
E2105: 'template' qualifier must specify a member template name (C++) 260
E2066: Information not available (C++) 261
E2471: pragma checkoption failed: options are not as expected (C++) 261
E2504: 'dynamic' can only be used with non-template member functions (C++) 261
E2191: '__far16' may only be used with '__pascal' or '__cdecl' (C++) 261
E2199: Template friend function 'function' must be previously declared (C++) 262

E2502: Error resolving #import: problem (C++) 262
E2501: Unable to open import file 'filename' (C++) 262
E2494: Unrecognized __declspec modifier (C++) 262
E2493: Invalid GUID string (C++) 263
E2499: Invalid __declspec(uuid(GuidString)) format (C++) 263
E2496: Invalid call to uuidof(struct type|variable) (C++) 263
E2511: Unterminated macro argument (C++) 263
E2489: Maximum option context replay depth exceeded; check for recursion (C++) 263
E2488: Maximum token reply depth exceeded; check for recursion (C++) 263
E2491: Maximum VIRDEF count exceeded; check for recursion (C++) 263
E2230: In-line data member initialization requires an integral constant expression (C++) 264
E2241: VCL style classes need virtual destructors (C++) 264
E2524: Anonymous structs/unions not allowed to have anonymous members in C++ (C++) 264
E2246: x is not abstract public single inheritance class hierarchy with no data (C++) 264
E2249: = expected (C++) 264
E2267: First base must be VCL class (C++) 264
E2472: Cannot declare a member function via instantiation (C++) 265
E2515: Cannot explicitly specialize a member of a generic template class (C++) 265
E2474: 'function' cannot be declared as static or inline (C++) 265
E2498: Need previously defined struct GUID (C++) 266
E2295: Too many candidate template specializations from 'specifier' (C++) 266
E2475: 'function' cannot be a template function (C++) 266
E2299: Cannot generate template specialization from 'specifier' (C++) 267
E2300: Could not generate a specialization matching type for 'specifier' (C++) 267
E2497: No GUID associated with type:'type' (C++) 267
E2522: Non-const function 'function' called for const object (C++) 267
E2523: Non-volatile function 'name' called for volatile object (C++) 267
E2513: Cannot emit RTTI for 'parameter' in 'function' (C++) 267
E2512: Cannot emit RTTI for return type of 'function' (C++) 267
E2507: 'class' is not a direct base class of 'class' (C++) 268
E2529: Path 'path' exceeds maximum size of 'n' (C++) 268
E2495: Redefinition of uuid is not identical (C++) 268
E2500: __declspec(selectany) is only for initialized and externally visible variables (C++) 268
E2482: String constant expected (C++) 268
E2481: Unexpected string constant (C++) 268
E2386: Cannot involve parameter 'parameter' in a complex partial specialization expression (C++) 268
E2387: Partial specializations may not specialize dependent non-type parameters ('parameter') (C++) 269
E2388: Argument list of specialization cannot be identical to the parameter list of primary template (C++) 269
E2389: Mismatch in kind of substitution argument and template parameter 'parameter' (C++) 269

E2480: Cannot involve template parameters in complex partial specialization arguments (C++) 269
E2392: Template instance 'template' is already instantiated (C++) 270
E2393: Cannot take the address of non-type, non-reference template parameter 'parameter' (C++) 270
E2399: Cannot reference template argument 'arg' in template class 'class' this way (C++) 270
E2397: Template argument cannot have static or local linkage (C++) 270
E2485: Cannot use address of array element as non-type template argument (C++) 271
E2402: Illegal base class type: formal type 'type' resolves to 'type' (C++) 271
E2403: Dependent call specifier yields non-function 'name' (C++) 271
E2404: Dependent type qualifier 'qualifier' has no member type named 'name' (C++) 271
E2405: Dependent template reference 'identifier' yields non-template symbol (C++) 271
E2406: Dependent type qualifier 'qualifier' is not a class or struct type (C++) 272
E2407: Dependent type qualifier 'qualifier' has no member symbol named 'name' (C++) 272
E2408: Default values may be specified only in primary class template declarations (C++) 272
E2409: Cannot find a valid specialization for 'specifier' (C++) 272
E2410: Missing template parameters for friend template 'template' (C++) 273
E2486: Cannot use address of class member as non-type template argument (C++) 273
E2411: Declaration of member function default parameters after a specialization has already been expanded (C++) 273
E2412: Attempting to bind a member reference to a dependent type (C++) 273
E2414: Destructors cannot be declared as template functions (C++) 273
E2473: Invalid explicit specialization of 'specifier' (C++) 274
E2490: Specialization within template classes not yet implemented (C++) 274
E2416: Invalid template function declaration (C++) 274
E2417: Cannot specify template parameters in explicit specialization of 'specifier' (C++) 274
E2418: Maximum instantiation depth exceeded; check for recursion (C++) 274
E2420: Explicit instantiation can only be used at global scope (C++) 274
E2422: Argument kind mismatch in redeclaration of template parameter 'parameter' (C++) 275
E2423: Explicit specialization or instantiation of non-existing template 'template' (C++) 275
E2479: Cannot have both a template class and function named 'name' (C++) 275
E2484: The name of template class 'class' cannot be overloaded (C++) 275
E2426: Explicit specialization of 'specifier' requires 'template<>' declaration (C++) 275
E2487: Cannot specify default function arguments for explicit specializations (C++) 276
E2427: 'main' cannot be a template function (C++) 276
E2429: Not a valid partial specialization of 'specifier' (C++) 276
E2430: Number of template parameters does not match in redeclaration of 'specifier' (C++) 276
E2477: Too few template parameters were declared for template 'template' (C++) 276
E2478: Too many template parameters were declared for template 'template' (C++) 277
E2431: Non-type template parameters cannot be of floating point, class, or void type (C++) 277
E2434: Template declaration missing template parameters ('template<...>') (C++) 277

E2435: Too many template parameter sets were specified (C++) 277
E2436: Default type for template template argument 'arg' does not name a primary template class (C++) 277
E2437: 'typename' should be followed by a qualified, dependent type name (C++) 278
E2438: Template template arguments must name a class (C++) 278
E2439: 'typename' is only allowed in template declarations (C++) 278
E2440: Cannot generate specialization from 'specifier' because that type is not yet defined (C++) 278
E2441: Instantiating 'specifier' (C++) 278
E2503: Missing or incorrect version of TypeLibImport.dll (C++) 279
E2470: Need to include header <typeinfo> to use typeid (C++) 279
E2514: Cannot (yet) use member overload resolution during template instantiation (C++) 279
E2508: 'using' cannot refer to a template specialization (C++) 279
E2462: 'virtual' can only be used with non-template member functions (C++) 279
W8086: Incorrect use of #pragma alias "aliasName"="substituteName" (C++) 280
W8099: Static main is not treated as an entry point (C++) 280
W8093: Incorrect use of #pragma codeseg [seg_name] ["seg_class"] [group] (C++) 280
W8094: Incorrect use of #pragma comment(<type> [,"string"]) (C++) 280
W8085: Function 'function' redefined as non-inline (C++) 281
W8105: %s member '%s' in class without constructors (C++) 281
W8095: Incorrect use of #pragma message("string") (C++) 281
W8098: Multi-character character constant (C++) 281
W8096: Incorrect use of #pragma code_seg(["seg_name"[,"seg_class"]]) (C++) 281
W8083: Pragma pack pop with no matching pack push (C++) 282
W8097: Not all options can be restored at this time (C++) 282
W8084: Suggest parentheses to clarify precedence (C++) 282
W8092: 'type' argument 'specifier' passed to 'function' is not an iterator: 'type' iterator required (C++) 282
W8087: 'operator::operator==' must be publicly visible to be contained by a 'type' (C++) 283
W8090: 'type::operator<' must be publicly visible to be used with 'type' (C++) 283
W8089: 'type::operator<' must be publicly visible to be contained by a 'type' (C++) 283
W8091: 'type' argument 'specifier' passed to 'function' is a 'iterator category' iterator: 'iterator category' iterator required (C++) 283
W8076: Template instance 'specifier' is already instantiated (C++) 283
W8077: Explicitly specializing an explicitly specialized class member makes no sense (C++) 283
Informational messages (C++) 284
E2196: Cannot take address of member function 'function' (C++) 284
F1002: Unable to create output file 'filename' (C++) 284
F1003: Error directive: 'message' (C++) 284
F1004: Internal compiler error (C++) 284
F1006: Bad call of intrinsic function (C++) 284
F1007: Irreducible expression tree (C++) 285

F1009: Unable to open input file 'filename' (C++) 285
F1011: Register allocation failure (C++) 285
F1012: Compiler stack overflow (C++) 285
F1013: Error writing output file (C++) 285
F1000: Compiler table limit exceeded (C++) 285
F1005: Include files nested too deep (C++) 286
F1008: Out of memory (C++) 286
F1010: Unable to open 'filename' (C++) 286
E2000: 286/287 instructions not enabled (C++) 286
Abnormal program termination (C++) 286
E2009: Attempt to grant or reduce access to 'identifier' (C++) 286
E2011: Illegal to take address of bit field (C++) 286
E2010: Cannot add or subtract relocatable symbols (C++) 287
E2013: 'function1' cannot be distinguished from 'function2' (C++) 287
E2014: Member is ambiguous: 'member1' and 'member2' (C++) 287
E2015: Ambiguity between 'function1' and 'function2' (C++) 287
E2017: Ambiguous member name 'name' (C++) 287
E2019: 'identifier' cannot be declared in an anonymous union (C++) 288
E2020: Global anonymous union not static (C++) 288
E2022: Array size too large (C++) 288
E2024: Cannot modify a const object (C++) 288
E2025: Assignment to 'this' not allowed, use X::operator new instead (C++) 288
E2026: Assembler statement too long (C++) 288
E2001: Constructors and destructors not allowed in __automated section (C++) 289
E2002: Only __fastcall functions allowed in __automated section (C++) 289
E2003: Data member definition not allowed in __automated section (C++) 289
E2004: Only read or write clause allowed in property declaration in __automated section (C++) 289
E2005: Redeclaration of property not allowed in __automated section (C++) 290
E2027: Must take address of a memory location (C++) 290
E2028: operator -> must return a pointer or a class (C++) 290
E2029: 'identifier' must be a previously defined class or struct (C++) 290
E2030: Misplaced break (C++) 290
E2031: Cannot cast from 'type1' to 'type2' (C++) 290
E2033: Misplaced continue (C++) 291
E2034: Cannot convert 'type1' to 'type2' (C++) 291
E2036: Conversion operator cannot have a return type specification (C++) 292
E2037: The constructor 'constructor' is not allowed (C++) 292
E2039: Misplaced decimal point (C++) 292
E2041: Incorrect use of default (C++) 292

E2042: Declare operator delete (void*) or (void*, size_t) (C++) 293
E2044: operator delete must return void (C++) 293
E2045: Destructor name must match the class name (C++) 293
E2048: Unknown preprocessor directive: 'identifier' (C++) 293
E2046: Bad file name format in include directive OR Bad file name format in line directive (C++) 294
E2051: Invalid use of dot (C++) 294
E2053: Misplaced elif directive (C++) 294
E2054: Misplaced else (C++) 294
E2055: Misplaced else directive (C++) 294
E2056: Misplaced endif directive (C++) 295
E2059: Unknown language, must be C or C++ (C++) 295
E2060: Illegal use of floating point (C++) 295
E2061: Friends must be functions or classes (C++) 295
E2062: Invalid indirection (C++) 295
E2063: Illegal initialization (C++) 296
E2064: Cannot initialize 'type1' with 'type2' (C++) 296
E2068: 'identifier' is not a non-static data member and can't be initialized here (C++) 296
E2069: Illegal use of member pointer (C++) 296
E2071: operator new must have an initial parameter of type size_t (C++) 297
E2072: Operator new[] must return an object of type void (C++) 297
E2075: Incorrect 'type' option: option (C++) 297
E2076: Overloadable operator expected (C++) 297
E2080: 'function' must be declared with one parameter (C++) 298
E2077: 'operator' must be declared with one or no parameters (C++) 298
E2079: 'function' must be declared with no parameters (C++) 298
E2078: 'operator' must be declared with one or two parameters (C++) 298
E2081: 'function' must be declared with two parameters (C++) 298
E2082: 'identifier' must be a member function or have a parameter of class type (C++) 298
E2083: Last parameter of 'operator' must have type 'int' (C++) 299
E2084: Parameter names are used only with a function body (C++) 299
E2085: Invalid pointer addition (C++) 299
E2086: Illegal pointer subtraction (C++) 299
E2087: Illegal use of pointer (C++) 299
E2088: Bad syntax for pure function definition (C++) 300
E2089: Identifier 'identifier' cannot have a type qualifier (C++) 300
E2090: Qualifier 'identifier' is not a class or namespace name (C++) 300
E2092: Storage class 'storage class' is not allowed here (C++) 300
E2096: Illegal structure operation (C++) 300
E2104: Invalid use of template keyword (C++) 300

E2108: Improper use of typedef 'identifier' (C++) 301
E2109: Not an allowed type (C++) 301
E2110: Incompatible type conversion (C++) 301
E2113: Virtual function 'function1' conflicts with base class 'base' (C++) 301
E2114: Multiple base classes require explicit class names (C++) 301
E2115: Bit field too large (C++) 301
E2116: Bit fields must contain at least one bit (C++) 301
W8005: Bit fields must be signed or unsigned int (C++) 302
E2119: User break (C++) 302
E2111: Type 'typename' may not be defined here (C++) 302
E2121: Function call missing) (C++) 302
E2123: Class 'class' may not contain pure functions (C++) 302
E2126: Case bypasses initialization of a local variable (C++) 302
E2127: Case statement missing : (C++) 302
E2128: Case outside of switch (C++) 302
E2129: Character constant too long (or empty) (C++) 303
E2133: Unable to execute command 'command' (C++) 303
E2134: Compound statement missing closing brace (C++) 303
E2137: Destructor for 'class' required in conditional expression (C++) 303
E2135: Constructor/Destructor cannot be declared 'const' or 'volatile' (C++) 303
E2138: Conflicting type modifiers (C++) 303
E2136: Constructor cannot have a return type specification (C++) 303
E2038: Cannot declare or define 'identifier' here: wrong namespace (C++) 304
E2154: Cannot define 'identifier' using a namespace alias (C++) 304
E2421: Cannot use local type 'identifier' as template argument (C++) 304
E2035: Conversions of class to itself or base class not allowed (C++) 304
E2139: Declaration missing ; (C++) 304
E2140: Declaration is not allowed here (C++) 304
E2141: Declaration syntax error (C++) 304
E2142: Base class 'class' contains dynamically dispatchable functions (C++) 304
E2143: Matching base class function 'function' has different dispatch number (C++) 305
E2144: Matching base class function 'function' is not dynamic (C++) 305
E2145: Functions 'function1' and 'function2' both use the same dispatch number (C++) 305
E2146: Need an identifier to declare (C++) 305
E2147: 'identifier' cannot start a parameter declaration (C++) 305
E2150: Type mismatch in default argument value (C++) 305
E2152: Default expression may not use local variables (C++) 306
E2153: Define directive needs an identifier (C++) 306
E2155: Too many default cases (C++) 306

E2156: Default outside of switch (C++) 306
E2158: Operand of 'delete' must be non-const pointer (C++) 306
E2159: Trying to derive a far class from the huge base 'base' (C++) 306
E2160: Trying to derive a far class from the near base 'base' (C++) 306
E2161: Trying to derive a huge class from the far base 'base' (C++) 307
E2162: Trying to derive a huge class from the near base 'base' (C++) 307
E2163: Trying to derive a near class from the far base 'base' (C++) 307
E2164: Trying to derive a near class from the huge base 'base' (C++) 307
E2165: Destructor cannot have a return type specification (C++) 307
E2166: Destructor for 'class' is not accessible (C++) 307
E2167: 'function' was previously declared with the language 'language' (C++) 307
E2168: Division by zero (C++) 308
E2169: 'identifier' specifies multiple or duplicate access (C++) 308
E2170: Base class 'class' is included more than once (C++) 308
E2171: Body has already been defined for function 'function' (C++) 308
E2172: Duplicate case (C++) 308
E2175: Too many storage classes in declaration (C++) 308
E2176: Too many types in declaration (C++) 309
E2179: virtual specified more than once (C++) 309
E2007: Dispid only allowed in __automated sections (C++) 309
Divide error (C++) 309
E2182: Illegal parameter to __emit__ (C++) 309
E2183: File must contain at least one external declaration (C++) 309
E2184: Enum syntax error (C++) 310
E2185: The value for 'identifier' is not within the range of 'type-name' (C++) 310
E2186: Unexpected end of file in comment started on line 'number' (C++) 310
E2187: Unexpected end of file in conditional started on line 'number' (C++) 310
E2188: Expression syntax (C++) 310
E2190: Unexpected closing brace (C++) 311
E2189: extern variable cannot be initialized (C++) 311
E2344: Earlier declaration of 'identifier' (C++) 311
E2192: Too few parameters in call (C++) 311
E2193: Too few parameters in call to 'function' (C++) 311
E2194: Could not find file 'filename' (C++) 311
E2197: File name too long (C++) 311
E2195: Cannot evaluate function call (C++) 312
E2198: Not a valid expression format type (C++) 312
E2200: Functions may not be part of a struct or union (C++) 312
Floating point error: Divide by 0 OR Floating point error: Domain OR Floating point error: Overflow (C++) 312

Floating point error: Stack fault (C++) 312
Floating point error: Partial loss of precision OR Floating point error: Underflow (C++) 312
E2201: Too much global data defined in file (C++) 313
E2203: Goto bypasses initialization of a local variable (C++) 313
E2204: Group overflowed maximum size: 'name' (C++) 313
E2206: Illegal character 'character' (0x'value') (C++) 313
E2207: Implicit conversion of 'type1' to 'type2' not allowed (C++) 313
E2208: Cannot access an inactive scope (C++) 313
E2209: Unable to open include file 'filename' (C++) 314
E2210: Reference member 'member' is not initialized (C++) 314
E2212: Function defined inline after use as extern (C++) 314
E2211: Inline assembly not allowed in inline and template functions (C++) 314
F1001: Internal code generator error (C++) 315
E2413: Invalid template declaration (C++) 315
E2070: Invalid use of namespace 'identifier' (C++) 315
E2214: Cannot have a 'non-inline function/static data' in a local class (C++) 315
E2215: Linkage specification not allowed (C++) 315
E2216: Unable to create turboc.$ln (C++) 315
E2218: Templates can only be declared at namespace or class scope (C++) 315
E2217: Local data exceeds segment size limit (C++) 316
E2219: Wrong number of arguments in call of macro 'macro' (C++) 316
E2220: Invalid macro argument separator (C++) 316
E2221: Macro argument syntax error (C++) 316
E2222: Macro expansion too long (C++) 316
E2223: Too many decimal points (C++) 316
E2224: Too many exponents (C++) 316
E2225: Too many initializers (C++) 316
E2226: Extra parameter in call (C++) 317
E2227: Extra parameter in call to function (C++) 317
E2228: Too many error or warning messages (C++) 317
E2233: Cannot initialize a class member here (C++) 317
E2232: Constant/Reference member 'member' in class without constructors (C++) 317
E2229: Member 'member' has the same name as its class (C++) 317
E2234: Memory reference expected (C++) 317
E2231: Member 'member' cannot be used without an object (C++) 318
E2235: Member function must be called or its address taken (C++) 318
O2237: DPMI programs must use the large memory model (C++) 318
E2238: Multiple declaration for 'identifier' (C++) 318
E2239: 'identifier' must be a member function (C++) 319

E2240: Conversion of near pointer not allowed (C++) 319
E2243: Array allocated using 'new' may not have an initializer (C++) 319
E2244: 'new' and 'delete' not supported (C++) 319
E2245: Cannot allocate a reference (C++) 319
E2309: Inline assembly not allowed (C++) 320
E2250: No base class to initialize (C++) 320
E2254: : expected after private/protected/private (C++) 320
E2255: Use :: to take the address of a member function (C++) 320
E2256: No : following the ? (C++) 320
E2257: , expected (C++) 320
E2258: Declaration was expected (C++) 320
E2259: Default value missing (C++) 321
E2260: Default value missing following parameter 'parameter' (C++) 321
E2263: Exception handling not enabled (C++) 321
E2264: Expression expected (C++) 321
E2266: No file names given (C++) 321
E2265: No file name ending (C++) 321
E2271: Goto statement missing label (C++) 321
E2272: Identifier expected (C++) 321
E2275: Opening brace expected (C++) 322
E2276: (expected (C++) 322
E2274: < expected (C++) 322
E2277: Lvalue required (C++) 322
E2278: Multiple base classes not supported for Delphi classes (C++) 322
E2280: Member identifier expected (C++) 322
E2279: Cannot find default constructor to initialize member 'identifier' (C++) 323
E2310: Only member functions may be 'const' or 'volatile' (C++) 323
E2311: Non-virtual function 'function' declared pure (C++) 323
E2283: Use . or -> to call 'function' (C++) 323
E2284: Use . or -> to call 'member', or & to take its address (C++) 323
E2285: Could not find a match for 'argument(s)' (C++) 323
E2286: Overloaded function resolution not supported (C++) 324
E2287: Parameter 'number' missing name (C++) 324
E2288: Pointer to structure required on left side of -> or ->* (C++) 324
E2290: 'code' missing] (C++) 324
E2291: brace expected (C++) 324
E2292: Function should return a value (C++) 324
E2293:) expected (C++) 325
E2294: Structure required on left side of . or .* (C++) 325

E2312: 'constructor' is not an unambiguous base class of 'class' (C++) 325
E2313: Constant expression required (C++) 325
E2296: Templates not supported (C++) 325
E2314: Call of nonfunction (C++) 325
E2321: Declaration does not specify a tag or an identifier (C++) 325
E2297: 'this' can only be used within a member function (C++) 326
E2316: 'identifier' is not a member of 'struct' (C++) 326
E2317: 'identifier' is not a parameter (C++) 326
E2319: 'identifier' is not a public base class of 'classtype' (C++) 326
E2320: Expression of scalar type expected (C++) 326
E2302: No type information (C++) 327
E2303: Type name expected (C++) 327
E2304: 'Constant/Reference' variable 'variable' must be initialized (C++) 327
E2305: Cannot find 'class::class' ('class'&) to copy a vector OR Cannot find 'class'::operator=('class'&) to copy a vector (C++) 327
E2306: Virtual base classes not supported for Delphi classes (C++) 328
E2308: do statement must have while (C++) 328
E2322: Incorrect number format (C++) 328
E2324: Numeric constant too large (C++) 328
E2282: Namespace name expected (C++) 328
E2334: Namespace member 'identifier' declared outside its namespace (C++) 328
E2325: Illegal octal digit (C++) 329
E2329: Invalid combination of opcode and operands (C++) 329
E2327: Operators may not have default argument values (C++) 329
E2330: Operator must be declared as function (C++) 329
E2333: Class member 'member' declared outside its class (C++) 329
E2335: Overloaded 'function name' ambiguous in this context (C++) 329
E2339: Cannot overload 'main' (C++) 330
E2336: Pointer to overloaded function 'function' doesn't match 'type' (C++) 330
E2337: Only one of a set of overloaded functions can be "C" (C++) 330
E2338: Overlays only supported in medium, large, and huge memory models (C++) 330
E2340: Type mismatch in parameter 'number' (C++) 330
E2341: Type mismatch in parameter 'number' in call to 'function' (C++) 331
E2342: Type mismatch in parameter 'parameter' (C++) 331
E2343: Type mismatch in parameter 'parameter' in call to 'function' (C++) 331
E2345: Access can only be changed to public or protected (C++) 331
E2349: Nonportable pointer conversion (C++) 331
E2350: Cannot define a pointer or reference to a reference (C++) 332
E2352: Cannot create instance of abstract class 'class' (C++) 332

E2354: Two operands must evaluate to the same type (C++) 332
E2355: Recursive template function: 'x' instantiated 'y' (C++) 333
E2356: Type mismatch in redeclaration of 'identifier' (C++) 333
E2357: Reference initialized with 'type1', needs lvalue of type 'type2' (C++) 333
E2358: Reference member 'member' needs a temporary for initialization (C++) 333
E2360: Invalid register combination (e.g. [BP+BX]) (C++) 334
E2361: 'specifier' has already been included (C++) 334
E2362: Repeat count needs an lvalue (C++) 334
E2363: Attempting to return a reference to local variable 'identifier' (C++) 334
E2364: Attempting to return a reference to a local object (C++) 334
E2365: Member pointer required on right side of .* or ->* (C++) 335
E2366: Can't inherit non-RTTI class from RTTI base OR E2367 Can't inherit RTTI class from non-RTTI base (C++) 335
E2368: RTTI not available for expression evaluation (C++) 335
E2371: sizeof may not be applied to a bit field (C++) 335
E2372: sizeof may not be applied to a function (C++) 335
E2373: Bit field cannot be static (C++) 335
E2374: Function 'function' cannot be static (C++) 335
Stack overflow (C++) 335
E2376: statement missing (C++) 336
E2377: statement missing) (C++) 336
E2378: do-while or for statement missing ; (C++) 336
E2379: Statement missing ; (C++) 336
E2380: Unterminated string or character constant (C++) 336
E2381: Structure size too large (C++) 336
E2382: Side effects are not allowed (C++) 336
E2383: Switch selection expression must be of integral type (C++) 336
E2433: Specialization after first use of template (C++) 337
E2384: Cannot call near class member function with a pointer of type 'type' (C++) 337
E2390: Type mismatch in parameter 'number' in template class name 'template' (C++) 337
E2391: Type mismatch in parameter 'parameter' in template class name 'template' (C++) 337
E2394: Too few arguments passed to template 'template' (C++) 337
E2395: Too many arguments passed to template 'template' (C++) 338
E2396: Template argument must be a constant expression (C++) 338
E2401: Invalid template argument list (C++) 338
E2400: Nontype template argument must be of scalar type (C++) 338
E2415: Template functions may only have 'type-arguments' (C++) 338
E2425: 'member' is not a valid template type member (C++) 338
E2428: Templates must be classes or functions (C++) 338

E2432: 'template' qualifier must name a template class or function instance' (C++) 339
E2442: Two consecutive dots (C++) 339
E2443: Base class 'class' is initialized more than once (C++) 339
E2444: Member 'member' is initialized more than once (C++) 339
E2445: Variable 'identifier' is initialized more than once (C++) 339
E2446: Function definition cannot be a typedef'ed declaration (C++) 339
E2132: Templates and overloaded operators cannot have C linkage (C++) 339
E2447: 'identifier' must be a previously defined enumeration tag (C++) 340
E2448: Undefined label 'identifier' (C++) 340
E2449: Size of 'identifier' is unknown or zero (C++) 340
E2450: Undefined structure 'structure' (C++) 340
E2451: Undefined symbol 'identifier' (C++) 340
E2453: Size of the type 'identifier' is unknown or zero (C++) 340
E2452: Size of the type is unknown or zero (C++) 341
E2454: union cannot be a base type (C++) 341
E2455: union cannot have a base type (C++) 341
E2456: Union member 'member' is of type class with 'constructor' (or destructor, or operator =) (C++) 341
E2461: '%s' requires run-time initialization/finalization (C++) 341
E2464: 'virtual' can only be used with member functions (C++) 341
E2465: unions cannot have virtual member functions (C++) 342
E2466: void & is not a valid type (C++) 342
E2467: 'Void function' cannot return a value (C++) 342
E2468: Value of type void is not allowed (C++) 342
E2469: Cannot use tiny or huge memory model with Windows (C++) 342
E2006: CodeGuarded programs must use the large memory model and be targeted for Windows (C++) 342
E2269: The function 'function' is not available (C++) 342
E2124: Invalid function call (C++) 343
E2213: Invalid 'expression' in scope override (C++) 343
E2236: Missing 'identifier' in scope override (C++) 343
Pure virtual function called (C++) 343
E2095: String literal not allowed in this context (C++) 343
Unexpected termination during compilation [Module Seg#:offset] OR Unexpected termination during linking [Module Seg#:offset] (C++) 343
E2012: Cannot take address of 'main' (C++) 344
E2016: Ambiguous override of virtual base member 'base_function': 'derived_function' (C++) 344
E2021: Array must have at least one element (C++) 344
E2023: Array of references is not allowed (C++) 344
E2032: Illegal use of closure pointer (C++) 344
E2040: Declaration terminated incorrectly (C++) 345

E2047: Bad 'directive' directive syntax (C++) 345
E2049: Class type 'type' cannot be marked as __declspec(delphireturn) (C++) 345
E2052: Dynamic function 'function' conflicts with base class 'class' (C++) 345
E2057: Exception specification not allowed here (C++) 345
E2058: Exception handling variable may not be used here (C++) 346
E2065: Using namespace symbol 'symbol' conflicts with intrinsic of the same name (C++) 346
E2067: 'main' must have a return type of int (C++) 346
E2073: Nothing allowed after pragma option pop (C++) 346
E2091: Functions cannot return arrays or functions (C++) 346
E2093: Operator 'operator' not implemented in type 'type' for arguments of the same type (C++) 346
E2094: Operator 'operator' not implemented in type 'type' for arguments of type 'type' (C++) 346
E2097: Explicit instantiation only allowed at file or namespace scope (C++) 347
E2098: Explicit specialization declarator "template<>" now required (C++) 347
E2099: Explicit specialization only allowed at file or namespace scope (C++) 347
E2101: 'export' keyword must precede a template declaration (C++) 347
E2103: Explicit instantiation must be used with a template class or function (C++) 347
E2106: Explicit specialization must be used with a template class or function (C++) 347
E2112: Unknown unit directive: 'directive' (C++) 347
E2118: Bit fields must have integral type (C++) 348
E2120: Cannot call 'main' from within the program (C++) 348
E2125: Compiler could not generate copy constructor for class 'class' OR Compiler could not generate default constructor for class 'class' OR Compiler could not generate operator = for class 'class' (C++) 348
E2130: Circular property definition (C++) 348
E2131: Objects of type 'type' cannot be initialized with { } (C++) 348
E2148: Default argument value redeclared for parameter 'parameter' (C++) 348
E2149: Default argument value redeclared (C++) 349
E2151: Type mismatch in default value for parameter 'parameter' (C++) 349
E2157: Deleting an object requires exactly one conversion to pointer operator (C++) 349
E2173: Duplicate handler for 'type1', already had 'type2' (C++) 349
E2174: The name handler must be last (C++) 349
E2177: Redeclaration of #pragma package with different arguments (C++) 350
E2178: VIRDEF name conflict for 'function' (C++) 350
E2180: Dispid number already used by identifier (C++) 350
E2181: Cannot override a 'dynamic/virtual' with a 'dynamic/virtual' function (C++) 350
E2202: Goto into an exception handler is not allowed (C++) 350
E2205: Illegal type type in __automated section (C++) 350
E2242: Specifier requires Delphi style class type (C++) 351
E2247: 'member' is not accessible (C++) 351
E2248: Cannot find default constructor to initialize array element of type 'class' (C++) 351

E2251: Cannot find default constructor to initialize base class 'class' (C++) 352
E2252: 'catch' expected (C++) 352
E2253: Calling convention must be attributed to the function type, not the closure (C++) 352
E2261: Use of dispid with a property requires a getter or setter (C++) 353
E2262: '__except' or '__finally' expected following '__try' (C++) 353
E2270: > expected (C++) 353
E2273: 'main' cannot be declared as static or inline (C++) 353
E2281: Identifier1 requires definition of Identifier2 as a pointer type (C++) 353
E2289: __published or __automated sections only supported for Delphi classes (C++) 353
E2298: Cannot generate 'function' from template function 'template' (C++) 354
E2301: Cannot use templates in closure arguments -- use a typedef (C++) 354
E2307: Type 'type' is not a defined class with virtual functions (C++) 354
E2315: 'Member' is not a member of 'class', because the type is not yet defined (C++) 354
E2318: 'type' is not a polymorphic class type (C++) 354
E2323: Illegal number suffix (C++) 355
E2326: Use __declspec(spec1[, spec2]) to combine multiple __declspecs (C++) 355
E2328: Classes with properties cannot be copied by value (C++) 355
E2331: Number of allowable option contexts exceeded (C++) 355
E2332: Variable 'variable' has been optimized and is not available (C++) 355
E2476: Cannot overload 'function' (C++) 356
E2346: 'x' access specifier of property 'property' must be a member function (C++) 356
E2347: Parameter mismatch in access specifier 'specifier' of property 'property' (C++) 356
E2348: Storage specifier not allowed for array properties (C++) 356
E2351: Static data members not allowed in __published or __automated sections (C++) 357
E2353: Class 'classname' is abstract because of 'member = 0' (C++) 357
E2359: Reference member 'member' initialized with a non-reference parameter (C++) 357
E2369: Cannot use the result of a property assignment as an rvalue' (C++) 358
E2370: Simple type name expected (C++) 358
E2398: Template function argument 'argument' not used in argument types (C++) 358
E2419: Error while instantiating template 'template' (C++) 358
E2424: Template class nesting too deep: 'class' (C++) 358
E2457: Delphi style classes must be caught by reference (C++) 359
E2458: Delphi classes have to be derived from Delphi classes (C++) 359
E2459: Delphi style classes must be constructed using operator new (C++) 359
E2460: Delphi style classes require exception handling to be enabled (C++) 360
E2463: 'base' is an indirect virtual base class of 'class' (C++) 360
Null pointer assignment (C++) 360
E2268: Call to undefined function 'function' (C++) 360
E2375: Assembler stack overflow (C++) 360

Initializing enumeration with type (C++) 360
<name> is not a valid identifier (C++) 361
Example for "Temporary used ..." error messages (C++) 361
Application is running (C++) 361
Printf/Scanf floating-point formats not linked (C++) 361
W8000: Ambiguous operators need parentheses (C++) 362
W8060: Possibly incorrect assignment (C++) 362
W8002: Restarting compile using assembly (C++) 362
W8003: Unknown assembler instruction (C++) 362
W8052: Base initialization without a class name is now obsolete (C++) 362
E2117: Bit fields must be signed or unsigned int (C++) 363
W8064: Call to function with no prototype (C++) 363
W8065: Call to function 'function' with no prototype (C++) 363
W8009: Constant is long (C++) 363
W8008: Condition is always true OR W8008 Condition is always false (C++) 363
W8012: Comparing signed and unsigned values (C++) 364
W8010: Continuation character \ found in // comment (C++) 364
W8080: 'identifier' is declared but never used (C++) 364
W8014: Declaration ignored (C++) 365
W8068: Constant out of range in comparison (C++) 365
W8016: Array size for 'delete' ignored (C++) 365
W8082: Division by zero (C++) 365
W8018: Assigning 'type' to 'enumeration' (C++) 365
W8006: Initializing 'identifier' with 'identifier' (C++) 366
W8001: Superfluous & with function (C++) 366
W8020: 'identifier' is declared as both external and static (C++) 366
W8007: Hexadecimal value contains more than three digits (C++) 366
W8024: Base class 'class1' is also a base class of 'class2' (C++) 366
W8022: 'function1' hides virtual function 'function2' (C++) 367
W8023: Array variable 'identifier' is near (C++) 367
W8061: Initialization is only partially bracketed (C++) 367
W8038: constant member 'identifier' is not initialized (C++) 367
W8071: Conversion may lose significant digits (C++) 367
W8043: Macro definition ignored (C++) 368
W8017: Redefinition of 'x' is not identical (C++) 368
W8079: Mixing pointers to different 'char' types (C++) 368
W8067: Both return and return with a value used (C++) 368
W8048: Use qualified name to access member type 'identifier' (C++) 368
W8039: Constructor initializer list ignored (C++) 368

W8040: Function body ignored (C++) 369
W8042: Initializer for object 'x' ignored (C++) 369
W8044: #undef directive ignored (C++) 369
W8037: Non-const function 'function' called for const object (C++) 369
W8051: Non-volatile function 'function' called for volatile object (C++) 369
W8019: Code has no effect (C++) 370
W8057: Parameter 'parameter' is never used (C++) 370
W8070: Function should return a value (C++) 370
W8047: Declaration of static function function ignored (C++) 370
W8041: Negating unsigned value (C++) 370
W8054: Style of function definition is now obsolete (C++) 371
W8025: Ill-formed pragma (C++) 371
W8063: Overloaded prefix operator 'operator' used as a postfix operator (C++) 371
W8015: Declare 'type' prior to use in prototype (C++) 371
W8069: Nonportable pointer conversion (C++) 372
W8066: Unreachable code (C++) 372
W8029: Temporary used for parameter '???' (C++) 372
W8031: Temporary used for parameter 'parameter' OR W8029 Temporary used for parameter 'number' OR W8030 Temporary used for parameter 'parameter' in call to 'function' OR W8032 Temporary used for parameter 'number' in call to 'function' (C++) 372
W8032: Temporary used for parameter 2 in call to '???' (C++) 373
W8028: Temporary used to initialize 'identifier' (C++) 373
W8074: Structure passed by value (C++) 373
W8011: Nonportable pointer comparison (C++) 374
W8075: Suspicious pointer conversion (C++) 374
W8059: Structure packing size has changed (C++) 374
W8045: No declaration for function 'function' (C++) 374
W8073: Undefined structure 'structure' (C++) 375
W8013: Possible use of 'identifier' before definition (C++) 375
W8004: 'identifier' is assigned a value that is never used (C++) 375
W8081: Void functions may not return a value (C++) 375
W8078: Throw expression violates exception specification (C++) 376
W8021: Handler for 'type1' hidden by previous handler for 'type2' (C++) 376
W8056: Integer arithmetic overflow (C++) 376
W8035: User-defined message (C++) 376
W8049: Use '> >' for nested templates Instead of '>>' (C++) 376
W8026: Functions with exception specifications are not expanded inline (C++) 377
W8058: Cannot create pre-compiled header: 'reason' (C++) 377
W8046: Pragma option pop with no matching option push (C++) 377
W8050: No type OBJ file present; disabling external types option. (C++) 377

W8027: Functions containing 'statement' are not expanded inline (C++) 378
W8036: Non-ANSI keyword used: 'keyword' (C++) 378
W8053: 'ident' is obsolete (C++) 379
W8103: Path 'path' and filename 'filename' exceed maximum size of 'n' (C++) 379
W8062: Previous options and warnings not restored (C++) 379
W8055: Possible overflow in shift operation (C++) 379
W8072: Suspicious pointer arithmetic (C++) 379
W8033: Conversion to 'type' will fail for members of virtual base 'class' (C++) 380
W8034: Maximum precision used for member pointer type 'type' (C++) 380
E2537: Cannot create instance of abstract class (C++) 380
E2018: Cannot catch 'identifier' -- ambiguous base class 'identifier' (C++) 380
E2550: No arguments can follow a variadic template in an argument list (C++) 381
E2538: Static assert failed: '%s' (C++) 381
E2548: ... was unexpected; expression is not a variadic template pattern (C++) 381
E2543: Combination of options 'options' is not permitted (C++) 381
E2549: Operand is not a parameter pack (C++) 381
E2544: Function exception specifications do not match (C++) 381
E2536: Incomplete type cannot be part of a exception declaration (C++) 381
E2535: Incomplete type cannot be part of a throw specification (C++) 382
E2532: Constant in new expression requires an initializer (C++) 382
E2541: Attribute '%s' cannot be set (C++) 382
E2545: Enum underlying type must be an integral (C++) 382
E2546: Redeclaration of enum is not identical (C++) 382
E2533: Parameter mismatch (wanted typename) (C++) 382
E2534: Integral constant expression expected (C++) 382
E2531: Parameter is an incomplete type (C++) 383
E2539: Constant expression expected (C++) 383
E2547: ... expected (C++) 383
E2540: String literal expected (C++) 383
E2552: This feature is not (yet) supported (C++) 383
E2542: '%s' is marked 'final' and cannot be overriden (C++) 383
E2553: %s mismatch in redeclaration of '%s' (C++) 383
E2551: Return statement not allowed in __finally block (C++) 384
W8104: Local Static with constructor dangerous for multi-threaded apps (C++) 384
W8106: %s are deprecated (C++) 384
W8110: Duplicate '%s' attribute directive ignored (C++) 384
W8108: Constant in new expression requires an initializer (C++) 384
W8113: Inline function was declared with 'extern template' (C++) 384
W8109: Parameter '%s' is a dependent type (C++) 385

W8105: Reference/Constant member 'identifier' in class without constructors (C++) 385
W8107: Type name expected (C++) 385
W8112: Unresolved dependencies in expression (C++) 385
C++ Language Guide 385
C++ Specifics 385
Keywords, Alphabetical Listing 513
Keywords, By Category 579
Language Structure 586
Lexical Elements 660
The Preprocessor 687
C Runtime Library Reference 711
alloc.h 712
assert.h 730
conio.h 732
ctype.h 766
delayimp.h 787
direct.h 790
dirent.h 792
dir.h 799
dos.h 818
errno.h 831
except.h 838
fastmath.h 842
fcntl.h 844
float.h 853
io.h 864
limits.h 913
locale.h 915
malloc.h 920
math.h 921
mem.h 959
new.h 968
process.h 971
setjmp.h 993
share.h 996
signal.h 997
stdarg.h 1003
stddef.h 1004
stdio.h 1006

stdlib.h 1080
string.h 1139
sys\stat.h 1186
sys\timeb.h 1190
sys\types.h 1192
time.h 1193
typeinfo.h 1211
utime.h 1213
values.h 1214

Win32 Developer's Guide 1217
Component Writer's Guide 1217
Creating a graphic component 1218
Creating events 1231
Creating methods 1241
Creating properties 1245
Customizing a grid 1258
Extending the IDE 1276
Handling messages 1298
Introduction to component creation 1310
Making a control data aware 1325
Making components available at design time 1339
Making a dialog box a component 1358
Modifying an existing component 1363
Object-oriented programming for component writers 1367
Using graphics in components 1376
Developing COM-based Applications 1381
COM basics 1382
Creating an Active Server Page 1400
Using ActiveX controls 1406
Creating COM clients 1418
Creating simple COM servers 1427
Working with type libraries 1441
Developing Database Applications 1469
Working with ADO components 1470
Connecting to databases 1495
Creating multi-tiered applications 1510
Creating reports with Rave Reports 1551
Designing database applications 1556
Understanding datasets 1573

Using the Borland Database Engine 1637
Using client datasets 1700
Using data controls 1743
Using decision support components 1779
Using provider components 1805
Using dbExpress Components 1821
Using XML in database applications 1840
Working with field components 1849
Programming with Delphi 1879
Building applications, components, and libraries 1879
Creating international applications 1920
Delphi programming fundamentals 1935
Deploying applications 1938
Developing the application user interface 1955
Exception handling 2014
Types of controls 2031
Understanding the component library 2053
Using the object model 2062
Using the VCL/RTL 2082
Working with components 2142
Working with controls 2149
Working with graphics and multimedia 2169
Working with packages and components 2208
Writing multi-threaded applications 2224
Writing Internet Applications 2243
Creating Internet server applications 2243
Using IntraWeb (VCL for the Web) 2254
Using Web Broker 2261
Using Web Services 2289
Using WebSnap 2310
Working with sockets 2335
Working with XML documents 2351

Index **a**

1 Concepts

Topics

Name	Description
Debugging C++ Applications with CodeGuard Error Reporting (see page 3)	CodeGuard provides runtime debugging for C++ applications developed with RAD Studio. CodeGuard reports errors that are not caught by the compiler because they do not violate syntax rules. CodeGuard tracks runtime libraries with full support for multithreaded applications.
Developing Database Applications for the Win32 Platform (see page 13)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress. Database applications let users interact with the information that is stored in the databases. Databases provide structure for the information, and allow it to be shared among different applications. Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as... more (see page 13)
Developing Interoperable Applications (see page 29)	RAD Studio provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms.
Developing Reports for Your Win32 Applications (see page 35)	RAD Studio ships with Rave Reports from Nevrona. Using the report components, you can build full-featured reports for your applications. You can create solutions that include reporting capabilities which can be used and customized by your customers. Additionally, the ComponentOne tools that ship with RAD Studio include components for creating and generating reports.
Developing Applications with VCL Components (see page 36)	The Visual Component Library (VCL) is a set of visual components for the rapid development of Windows applications in the Delphi language. VCL contains a wide variety of visual, non-visual, and utility classes for tasks such as building Windows applications, web applications, database applications, and console applications.
Developing Web Applications with WebSnap (see page 41)	This section provides a conceptual background for building WebSnap applications using RAD Studio. WebSnap makes it easier to build Web server applications that deliver complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripting makes development and maintenance easier for teams of developers and Web designers. Please note that WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL... more (see page 41)

Developing Web Services with Win32 Applications (see page 44)	Web Services are self-contained modular applications that can be published and invoked over the Internet. Web Services provide well-defined interfaces that describe the services provided. Unlike Web server applications that generate Web pages for client browsers, Web Services are not designed for direct human interaction. Rather, they are accessed programmatically by client applications. This section gives an overview of web services and web services support.
Developing Windows Applications (see page 45)	Windows provides a traditional approach to developing user interfaces, client/server applications, controls, and application logic. This section provides an overview of Windows application development using RAD Studio for Win32 and outlines the steps you would use to build a simple Windows project.

1.1 Debugging C++ Applications with CodeGuard Error Reporting

CodeGuard provides runtime debugging for C++ applications developed with RAD Studio. CodeGuard reports errors that are not caught by the compiler because they do not violate syntax rules. CodeGuard tracks runtime libraries with full support for multithreaded applications.

Topics

Name	Description
CodeGuard Errors (see page 3)	CodeGuard reports four types of runtime errors.
CodeGuard Overview (see page 9)	CodeGuard provides runtime debugging for C++ applications developed with RAD Studio. CodeGuard reports errors that are not caught by the compiler because they do not violate syntax rules. CodeGuard tracks runtime libraries with full support for multithreaded applications. CodeGuard provides two principal types of coverage: • Memory and Resource Use • Function Call Validation
CodeGuard Warnings (see page 10)	CodeGuard can report situations where your application may access memory beyond a buffer's maximum size. Warnings are available for three types of runtime library functions.

1.1.1 CodeGuard Errors

CodeGuard reports four types of runtime errors.

Topics

Name	Description
Access Errors (see page 4)	Access errors result from improper memory management. When CodeGuard detects accesses to freed memory blocks or deleted objects, it can identify where each block was allocated and deleted. Enable the **Delay Free** option using the **CodeGuard Configuration** dialog box to use this feature. The following are types of access errors: • Access in freed memory • Access in uninitialized stack • Access in invalid stack
Exception Errors (see page 5)	When a system exception occurs, CodeGuard reports the runtime error using information provided by the operating system. If possible, the CodeGuard log shows where your application caused the exception. CodeGuard does not trap or redirect the exception or otherwise interfere with normal program behavior. The following exceptions illustrate how CodeGuard exception reporting: • General Protection Fault • Divide by zero
Function Failure Errors (see page 6)	CodeGuard reports function calls that fail, as indicated by their return value. In the following example, the `close` function is given an invalid file handle, which causes it to return a value indicating that it was unable to close a file.

Resource Errors (see page 7)	Resources are memory blocks (allocated with functions like `malloc`, `GlobalAlloc`) and object arrays, such as file handles, stream handles, modules, and items returned by **new[]**. The following runtime error examples illustrate how CodeGuard reports improper use of resources: • Bad parameter • Reference to freed resource • Resource type mismatch • Resource leaks • Resource from different RTL

1.1.1.1 Access Errors

Access errors result from improper memory management.

When CodeGuard detects accesses to freed memory blocks or deleted objects, it can identify where each block was allocated and deleted. Enable the **Delay Free** option using the **CodeGuard Configuration** dialog box to use this feature.

The following are types of access errors:

- Access in freed memory
- Access in uninitialized stack
- Access in invalid stack

Access In Freed Memory

In the following example, CodeGuard identifies the line where an invalid access occurrs. CodeGuard then indicates where the memory block was allocated and subsequently freed.

```
Error 00004. 0x100430 (Thread 0xFFF87283):
Access in freed memory: Attempt to access 19 byte(s) at 0x00B423DC.
strcpy(0x00B423DC, 0x004091CA ["Copy to free block"])
| lang.cpp line 106:
|
|    free(buf_h);
|>   strcpy(buf_h, "Copy to free block");
|
|    //----------------------//
Call Tree:
   0x004011F1(=LANG.EXE:0x01:0001F1) lang.cpp#106
   0x00407EE5(=LANG.EXE:0x01:006EE5)

The memory block (0x00B423DC) [size: 21 bytes] was allocated with malloc
| lang.cpp line 80:
|    char * pad = (char *) malloc(200);
|    // An array in the RTL heap.
|>   char * buf_h = (char *) malloc(21);
|    char * p;
|    // A scratch buffer.
Call Tree:
   0x004011A1(=LANG.EXE:0x01:0001A1) lang.cpp#80
   0x00407EE5(=LANG.EXE:0x01:006EE5)

The memory block (0x00B423DC) was freed with free
| lang.cpp line 105:
|    //-------------//
|
|>   free(buf_h);
|    strcpy(buf_h, "Copy to free block");
```

```
|
Call Tree:
   0x004011E5(=LANG.EXE:0x01:0001E5) lang.cpp#105
   0x00407EE5(=LANG.EXE:0x01:006EE5)
```

Access In Uninitialized Stack

In the following example, the pointer `p` became invalid when `getBadLocal` returned from execution. No additional information is provided because the stack frame for `getBadLocal` was automatically removed.

```
Error 00005. 0x120400 (Thread 0xFFF87283):
Access in uninitialized stack: Attempt to access 20 byte(s) at 0x0072FC88.
memcpy(0x0072FCC4, 0x0072FC88, 0x14 [20])
| lang.cpp line 112:
|    //----------------------//
|    p = getBadLocal();
|>   memcpy(buffer, p, 20);
|
|    //-------------//
Call Tree:
   0x00401208(=LANG.EXE:0x01:000208) lang.cpp#112
   0x00407EE5(=LANG.EXE:0x01:006EE5)
```

Access In Invalid Stack

In the following example, an allocation was made for `buf_s` on the stack. However, the `strcpy` function writes just below the beginning of the valid stack region. CodeGuard identifies this as an error even if the string is only one byte long.

```
Error 00002. 0x110400 (Thread 0xFFF87283):
Access in invalid stack: Attempt to access 22 byte(s) at 0x0072FD8F.
strcpy(0x0072FD8F, 0x00409188 ["This string is long!\n"])
| LANG.CPP line 93:
|
|    // Stack underrun:
|>   strcpy(buf_s -1, "This string is long!\n");
|
|    // Global data overrun:
Call Tree:
   0x004011C5(=LANG.EXE:0x01:0001C5) LANG.CPP#93
   0x00407EED(=LANG.EXE:0x01:006EED)
```

1.1.1.2 Exception Errors

When a system exception occurs, CodeGuard reports the runtime error using information provided by the operating system. If possible, the CodeGuard log shows where your application caused the exception. CodeGuard does not trap or redirect the exception or otherwise interfere with normal program behavior.

The following exceptions illustrate how CodeGuard exception reporting:

- General Protection Fault
- Divide by zero

General Protection Fault

In the following example, CodeGuard provides information on a general protection fault (Intel system exception `0xD`). In addition to the location of the source code that caused the exception, the log shows where the memory was allocated and subsequently freed. The reported incorrect value is a result of accessing a byte pattern that CodeGuard uses to identify invalid memory locations.

```
Error 00003. 0x400003 (Thread 0x0090):
Exception 0xC0000005: Access violation at 0x80828082.
| gpfault.c line 32:
|    {¬
|       q = p[3];
```

```
|>      *q = 1;
|    }
| }
Call Tree:
   0x004010E5(=GPFAULT.EXE:0x01:0000E5) gpfault.c#32
   0x00406B29(=GPFAULT.EXE:0x01:005B29)

The bogus value (0x80828082) was most likely retrieved by accessing a(n)
 memory block that has already been freed
The memory block (0x008322A4) [size: 16 bytes] was allocated with malloc
| gpfault.c line 17:
|     int *q;
|
|>   p = malloc(sizeof(*p) * 4);
|
|    /*  Initialize p */
Call Tree:
   0x00401094(=GPFAULT.EXE:0x01:000094) gpfault.c#17
   0x00406B29(=GPFAULT.EXE:0x01:005B29)

The memory block (0x008322A4) was freed with free
| gpfault.c line 17:
|     int *q;
|
|>   p = malloc(sizeof(*p) * 4);
|
|    /*  Initialize p */
Call Tree:
   0x00401094(=GPFAULT.EXE:0x01:000094) gpfault.c#17
   0x00406B29(=GPFAULT.EXE:0x01:005B29)
```

Divide By Zero

In the following example, CodeGuard identifies the location in source code where division by zero (Intel system exception `0x0`) occurred.

```
Error 00001. 0x400000 (Thread 0x008B):
Exception 0xC0000094:
| ZERODIV.C line 9:
| {¬
|  x = 1;
|> return x / y;
| }
|
Call Tree:
   0x0040109C(=ZERODIV.EXE:0x01:00009C) ZERODIV.C#9
   0x00406321(=ZERODIV.EXE:0x01:005321)
```

1.1.1.3 Function Failure Errors

CodeGuard reports function calls that fail, as indicated by their return value.

In the following example, the `close` function is given an invalid file handle, which causes it to return a value indicating that it was unable to close a file.

```
Error 00009. 0x820000 (r) (Thread 0xFFF840F1):
Function failure:
close(0x80868086 [-2138668922])=0xFFFFFFFF [-1]
| lang.cpp line 125:
|    // uninitialized data usage //
|    //--------------------------//
|>   close(m->handle);
|
|
Call Tree:
```

```
   0x00401236(=LANG.EXE:0x01:000236) lang.cpp#125
   0x00407EED(=LANG.EXE:0x01:006EED)
```

1.1.1.4 Resource Errors

Resources are memory blocks (allocated with functions like `malloc`, `GlobalAlloc`) and object arrays, such as file handles, stream handles, modules, and items returned by **new[]**.

The following runtime error examples illustrate how CodeGuard reports improper use of resources:

- Bad parameter
- Reference to freed resource
- Resource type mismatch
- Resource leaks
- Resource from different RTL

Bad Parameter

When a resource is passed to a function, CodeGuard checks the runtime arguments. CodeGuard notifies you if it detects a bad parameter.

```
Error 00017. 0x310000 (Thread 0xFFF87283):
Bad parameter: A bad file handle (0xEA) has been passed to the function.
close(0xEA [234])
| lang.cpp line 170:
|    // using a bad handle //
|    //--------------------//
|>   close(234);
|
|    //----------------------//
Call Tree:
   0x00401456(=LANG.EXE:0x01:000456) lang.cpp#170
   0x00407EE5(=LANG.EXE:0x01:006EE5)
```

Reference To Freed Resource

In the following example, CodeGuard reports an attempt to read from a file that has already been closed. The CodeGuard log shows where the file was opened and subsequently closed.

```
Error 00020. 0x310030 (Thread 0xFFF840F1):
Reference to freed resource:
read(0x3 [3], 0x0072FCC4, 0x5 [5])
| lang.cpp line 177:
|    int i = open("lang.cpp", 0);
|    close(i);
|>   read (i, buffer, 5);
|
|    //--------------//
Call Tree:
   0x00401487(=LANG.EXE:0x01:000487) lang.cpp#177
   0x00407EED(=LANG.EXE:0x01:006EED)

The file handle (0x00000003) [name: 'lang.cpp'] was opened with open
| lang.cpp line 175:
|    // using a freed handle //
|    //----------------------//
|>   int i = open("lang.cpp", 0);
|    close(i);
|    read (i, buffer, 5);
Call Tree:
   0x0040146C(=LANG.EXE:0x01:00046C) lang.cpp#175
   0x00407EED(=LANG.EXE:0x01:006EED)
```

```
The file handle (0x00000003) was closed with close
| lang.cpp line 176:
|    //----------------------//
|    int i = open("lang.cpp", 0);
|>   close(i);
|    read (i, buffer, 5);
|
Call Tree:
   0x00401477(=LANG.EXE:0x01:000477) lang.cpp#176
   0x00407EED(=LANG.EXE:0x01:006EED)
```

Resource Type Mismatch

In the following example, a memory block that was allocated with the **new[]** operator, and should therefore be released with the **delete[]** operator, is instead released with a call to the `free` function.

```
Error 00024. 0x350010 (Thread 0xFFF840F1):
Resource type mismatch: a(n) memory block was expected.
free(0x00B42464)
| lang.cpp line 188:
|    //---------------//
|    char * ss = new char[21];
|>   free(ss);
|
| #ifdef    __WIN32__
Call Tree:
   0x0040149F(=LANG.EXE:0x01:00049F) lang.cpp#188
   0x00407EED(=LANG.EXE:0x01:006EED)

The object array (0x00B42464) [size: 21 bytes] was created with new[]
| lang.cpp line 187:
|    // type mismatch //
|    //---------------//
|>   char * ss = new char[21];
|    free(ss);
|
Call Tree:
   0x00401498(=LANG.EXE:0x01:000498) lang.cpp#187
   0x00407EED(=LANG.EXE:0x01:006EED)
```

Resource Leaks

In the following example, memory has been allocated but is never freed.

```
The memory block (0x00B42310) [size: 200 bytes] was allocated with malloc
| lang.cpp line 78:
|    // An array on the stack.
|    char buf_s[21];
|>   char * pad = (char *) malloc(200);
|    // An array in the RTL heap.
|    char * buf_h = (char *) malloc(21);
Call Tree:
   0x00401199(=LANG.EXE:0x01:000199) lang.cpp#78
   0x00407EE5(=LANG.EXE:0x01:006EE5)
```

Resource From Different RTL

CodeGuard reports an error if your application allocates, uses, or releases resources in different versions of the runtime library. This can happen, as the following example illustrates, if you link with a static runtime library but call a DLL.

Note: CodeGuard detects resource type mismatches before it detects mixed versions of the RTL. When the two kinds of error are combined, CodeGuard will not report the mixed RTLs until you correct the resource type mismatch.

```
Error 00001. 0x340010 (Thread 0x0062):
Resource from different RTL:
```

```
close(0x3 [3])
| testdll.cpp line 23:
| {¬
|    MessageBox(NULL,"RTLMixHandle: DLL closing EXE handle", "TESTDLL.CPP", MB_OK );
|>   close( handle );
|    return 1;
| }
Call Tree:
   0x0032115A(=testdll.dll:0x01:00015A) testdll.cpp#23
   0x00401660(=WINAPI.EXE:0x01:000660) filescg.cpp#33
   0x00401271(=WINAPI.EXE:0x01:000271) winapi.cpp#122
   0x77EA15B3
   0x00408B9A(=WINAPI.EXE:0x01:007B9A)

The file handle (0x00000003) [name: 'test2.dat'] was opened with open
| filescg.cpp line 32:
|
|  MessageBox(NULL,"FilesMixCG: Mixing RTL file handles", "FILESCG.CPP", MB_OK );
|> i = open("test2.dat", O_CREAT, S_IREAD | S_IWRITE );
|  RTLMixHandle( i );
| }
Call Tree:
   0x00401657(=WINAPI.EXE:0x01:000657) filescg.cpp#32
   0x00401271(=WINAPI.EXE:0x01:000271) winapi.cpp#122
   0x77EA15B3
   0x00408B9A(=WINAPI.EXE:0x01:007B9A)
```

1.1.2 CodeGuard Overview

CodeGuard provides runtime debugging for C++ applications developed with RAD Studio. CodeGuard reports errors that are not caught by the compiler because they do not violate syntax rules. CodeGuard tracks runtime libraries with full support for multithreaded applications.

CodeGuard provides two principal types of coverage:

- Memory and Resource Use
- Function Call Validation

Memory and Resource Use

CodeGuard checks for faulty memory use, improper memory allocation or deallocation, invalid file streams or handles, and resource leaks caused by improper use of file streams or handles. CodeGuard verifies pointer dereferencing and pointer arithmetic. CodeGuard can report an error if your program tries to access memory or resources that have already been released.

Function Call Validation

CodeGuard verifies function arguments and reports function failure as indicated by the return value of the function. It validates Windows resource handles used in function calls.

See Also

CodeGuard Errors (see page 3)

CodeGuard Warnings (see page 10)

Using CodeGuard (see page 48)

1

1.1.3 CodeGuard Warnings

CodeGuard can report situations where your application may access memory beyond a buffer's maximum size. Warnings are available for three types of runtime library functions.

Topics

Name	Description
Memory Block Comparison Warnings (see page 11)	Each of the following functions has a parameter that determines the maximum number of bytes it compares: 1. `memcmp` 2. `memicmp` 3. `_fmemcmp` 4. `_fmemicmp` If the **Warnings** option is enabled for the functions listed above, CodeGuard verifies that a comparison can be performed for each memory block passed to the function. If a memory block is too large, as determined by the parameter passed to the function, CodeGuard generates a warning. If the **Warnings** option is disabled for the functions listed above, CodeGuard checks the first byte in each memory block passed to the function. If the memory block is invalid, CodeGuard generates... more (see page 11)
Pathname Merging and Splitting Warnings (see page 11)	Each of the following functions use constants defined in `dir.h` to determine the maximum number of bytes to copy to or from a buffer: 1. `fnmerge` 2. `fnsplit` 3. `getcurdir`
String Comparison Warnings (see page 11)	Each of the following functions has a parameter that determines the maximum number of bytes it compares: • `strncmp` • `strnicmp` • `strncmpi` • `_fstrncmp` • `_fstrnicmp` If the **Warnings** option is enabled for the functions listed, CodeGuard verifies that a string comparison can be performed for each buffer passed to the function. If the buffer size is too large, as determined by the parameter passed to the function, and the buffer is not null-terminated, CodeGuard generates a warning. If the **Warnings** option is disabled for the functions listed above, CodeGuard checks the first byte in each memory block passed to the function. If the... more (see page 11)

1.1.3.1 Memory Block Comparison Warnings

Each of the following functions has a parameter that determines the maximum number of bytes it compares:

1. `memcmp`
2. `memicmp`
3. `_fmemcmp`
4. `_fmemicmp`

If the **Warnings** option is enabled for the functions listed above, CodeGuard verifies that a comparison can be performed for each memory block passed to the function. If a memory block is too large, as determined by the parameter passed to the function, CodeGuard generates a warning.

If the **Warnings** option is disabled for the functions listed above, CodeGuard checks the first byte in each memory block passed to the function. If the memory block is invalid, CodeGuard generates an error message.

1.1.3.2 Pathname Merging and Splitting Warnings

Each of the following functions use constants defined in `dir.h` to determine the maximum number of bytes to copy to or from a buffer:

1. `fnmerge`
2. `fnsplit`
3. `getcurdir`

fnmerge

If the **Warnings** option is enabled, the output buffer is validated against `MAXPATH` before `fnmerge` is called.

If the **Warnings** option is disabled, the size of the output buffer is validated against the null-terminated string length after `fnmerge` is called.

fnsplit

If the **Warnings** option is enabled, the input buffers are validated against `MAXDRIVE`, `MAXDIR`, `MAXFILE`, and `MAXEXT` before `fnsplit` is called.

If the **Warnings** option is disabled, the input buffers are validated against the length of the null-terminated string after `fnsplit` is called.

getcurdir

If the **Warnings** option is enabled, the output buffer is validated against `MAXDIR` before `getcurdir` is called.

If the **Warnings** option is disabled, the output buffer is validated against the length of the null-terminated string after `getcurdir` is called.

1.1.3.3 String Comparison Warnings

Each of the following functions has a parameter that determines the maximum number of bytes it compares:

- `strncmp`
- `strnicmp`
- `strncmpi`

- `_fstrncmp`
- `_fstrnicmp`

If the **Warnings** option is enabled for the functions listed, CodeGuard verifies that a string comparison can be performed for each buffer passed to the function. If the buffer size is too large, as determined by the parameter passed to the function, and the buffer is not null-terminated, CodeGuard generates a warning.

If the **Warnings** option is disabled for the functions listed above, CodeGuard checks the first byte in each memory block passed to the function. If the memory block is invalid, CodeGuard generates an error message.

1.2 Developing Database Applications for the Win32 Platform

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Database applications let users interact with the information that is stored in the databases. Databases provide structure for the information, and allow it to be shared among different applications.

Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

Topics

Name	Description
dbGo Overview (see page 14)	dbGo provides the developers with a powerful and logical object model for programmatically accessing, editing, and updating data from a wide variety of data sources through OLE DB system interfaces. The most common usage of dbGo is to query a table or tables in a relational database, retrieve and display the results in an application, and perhaps allow users to make and save changes to the data. The ADO layer of an ADO-based application consists of the latest version of Microsoft ADO, an OLE DB provider or ODBC driver for the data store access, client software for the... more (see page 14)
BDE Overview (see page 15)	**Warning:** The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress. The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers... more (see page 15)
dbExpress Components (see page 16)	dbExpress is a set of lightweight database components that provide fast access to SQL database servers. For each supported database, dbExpress provides a driver framework that adapts the server-specific software to a set of uniform dbExpress interfaces. When you deploy a database application that uses dbExpress, you include a DLL(the server-specific driver) with the application files you build. dbExpress lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command returns a set... more (see page 16)
Getting Started with InterBase Express (see page 17)	InterBase Express (IBX) is a set of data access components that provide a means of accessing data from InterBase databases. The InterBase Administration Components, which require InterBase, are described after the InterBase data access components.

dbExpress 4 Feature Overview (see page 22)	dbExpress's top level framework and metadata support has been rewritten in Delphi for RAD Studio 2007. It has new, richer metadata support. The DbxClient driver remotes the dbExpress 4 framework interface over a network based transport. This document discusses the following features: • dbExpress Framework • dbExpress Metadata Improvements • DBXClient Driver • DBXDynalink Driver • DBTest
Blackfish SQL Overview (see page 24)	The design and implementation of Blackfish SQL emphasizes database performance, scalability, ease of use, and a strong adherence to industry standards. Blackfish SQL capabilities include the following: • Industry standards compliance • Entry level SQL-92 • Unicode storage of character data • Unicode-based collation key support for sorting and indexing • dbExpress 4 drivers for win32 Delphi and C++ • ADO.NET 2.0 providers for .NET • JDBC for Java • JavaBean data access components for Java • XA/JTA Distributed transactions for Java • High performance and scalability for demanding online transaction processing (OLTP) and decision support system (DSS) applications • Delphi, C#, and VB.NET stored procedures and triggers... more (see page 24)
dbExpress Framework (see page 26)	The dbExpress framework (DBX framework) is a set of abstract classes provided in the unit DBXCommon. Applications can interface with the framework in several ways: using the framework directly for both native and managed applications, and using the dbExpress VCL components that are layered on top of the framework for both native and managed applications. Although many applications interface with dbExpress drivers via the dbExpress VCL components, the DBX framework offers a convenient, lighter weight option to communicate with a database driver. You can also create a database driver for dbExpress by extending the frameworks's DBXCommon abstract... more (see page 26)
dbExpress Framework Compatibility (see page 27)	Some dbExpress software developed prior to the dbExpress driver framework (DBX driver framework) has been modified to work with the DBX driver framework. As a result of these changes, some compatibility issues arise.

1.2.1 dbGo Overview

dbGo provides the developers with a powerful and logical object model for programmatically accessing, editing, and updating data from a wide variety of data sources through OLE DB system interfaces. The most common usage of dbGo is to query a table or tables in a relational database, retrieve and display the results in an application, and perhaps allow users to make and save changes to the data.

The ADO layer of an ADO-based application consists of the latest version of Microsoft ADO, an OLE DB provider or ODBC

driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these must be accessible to the ADO-based application for it to be fully functional.

The **dbGo** category of the **Tool Palette** hosts the **dbGo** components. These components let you connect to an ADO data store, execute commands, and retrieve data from tables in databases using the ADO framework. The components require the latest version of ADO to be installed on the host computer. Additionally, client software for the target database system (such as Microsoft SQL Server) must be installed, as well as an OLE DB driver or ODBC driver specific to the particular database system.

Most **dbGo** components have direct counterparts in the components available for other data access mechanisms: a database connection component, **TADOConnection**, and various types of datasets. In addition, **dbGo** includes **TADOCommand**, a simple component that is not a dataset but which represents an SQL command to be executed on the ADO data store.

The main **dbGo** components are:

Components	Function
TADOConnection	A database connection component that establishes a connection with an ADO data store. Multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and operate on metadata.
TRDSConnection	A database connection component to marshal data in multi-tier database applications that are built using ADO-based application servers.
TADODataSet	Primary dataset used for retrieving and operating on data. **TADODataSet** can retrieve data from a single or multiple tables, can connect directly to a data store, or use a **TADOConnection** component
TADOTable	A table-type dataset for retrieving and operating on a recordset produced by a single database table. **TADOTable** can connect directly to a data store or use a **TADOConnection** component
TADOQuery	A query-type dataset for retrieving and operating on a recordset produced by a valid SQL statement. **TADOQuery** can also execute Data Definition Language (DDL) SQL statements. It can connect directly to a data store or use a **TADOConnection** component.
TADOStoredProc	A stored procedure-type dataset for executing stored procedures. **TADOStoredProc** executes stored procedures that may or may not retrieve data. It can connect directly to a data store or use a **TADOConnection** component.
TADOCommand	A simple component for executing commands (SQL statements that do not return result sets). **TADOCommand** can be used with a supporting dataset component, or retrieve a dataset from a table. It can connect directly to a data store or use a **TADOConnection** component

See Also

Working with dbGo Components (see page 1494)

1.2.2 BDE Overview

Warning: The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers to connect to different databases. The components on the **BDE** category of the **Tool Palette** enable you to connect to database information

using the BDE.

When deploying BDE-based applications, you must include the BDE with your application. While this increases the size of the application and the complexity of deployment, the BDE can be shared with other BDE-based applications and provides a broader range of support for database manipulation. Although you can use the API of the BDE directly in your application, the components on the **BDE** category of the **Tool Palette** wrap most of this functionality for you.

The main **BDE** components are:

Components	Function
TTable	Retrieves data from a physical database table via the BDE and supplies it to one or more data-aware components through a DataSource component. Conversely, it also sends data received from a component to a physical database via the BDE.
TQuery	Uses SQL statements to retrieve data from a physical database table via the BDE and supplies it to one or more data-aware components through a TDataSource component. Conversely, it uses SQL statements to send data from a component to a physical database via the BDE.
TStoredProc	Enables an application to access server stored procedures. It sends data received from a component to a physical database via the BDE.
TDatabase	Sets up a persistent connection to a database, especially a remote database requiring a user login and password.
TSession	Provides global control over a group of database components. A default TSession component is automatically created for each database application. You must use the TSession component only if you are creating a multithreaded database application. Each database thread requires its own session component.
TBatchMove	Copies a table structure or its data. It can be used to move entire tables from one database format to another.
TUpdateSQL	Lets you use cached updates support with read-only datasets.
TNestedTable	Retrieves the data in a nested dataset field and supplies it to data-aware controls through a datasource component.

See Also

Using BDE (see page 60)

Borland Database Engine (see page 1946)

1.2.3 dbExpress Components

dbExpress is a set of lightweight database components that provide fast access to SQL database servers. For each supported database, dbExpress provides a driver framework that adapts the server-specific software to a set of uniform dbExpress interfaces. When you deploy a database application that uses dbExpress, you include a DLL(the server-specific driver) with the application files you build.

dbExpress lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command returns a set of records, retrieve those records. Unidirectional datasets do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets.

dbExpress connections, tables, views, and stored procedures that show up in a data tree view support drag & drop with native and managed vcl forms.

The **dbExpress** category of the **Tool Palette** contains components that use **dbExpress** to access database information. They are:

Components	Function
TSQLConnection	Encapsulates a dbExpress connection to a database server
TSQLDataSet	Represents any data available through **dbExpress**, or sends commands to a database accessed through **dbExpress**
TSQLQuery	A query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records, if any
TSQLTable	A table-type dataset that represents all of the rows and columns of a single database table
TSQLStoredProc	A stored procedure-type dataset that executes a stored procedure defined on a database server
TSQLMonitor	Intercepts messages that pass between an SQL connection component and a database server and saves them in a string list
TSimpleDataSet	A client dataset that uses an internal **TSQLDataSet** and **TDataSetProvider** for fetching data and applying updates

See Also

Using dbExpress (see page 72)

Using dbExpress Datasets (see page 1823)

Configuring TSQL Connection (see page 53)

Using Data Explorer to get Connection Information (see page 52)

1.2.4 Getting Started with InterBase Express

InterBase Express (IBX) is a set of data access components that provide a means of accessing data from InterBase databases. The InterBase Administration Components, which require InterBase, are described after the InterBase data access components.

IBX components

The following components are located on the InterBase tab of the component palette.

	TIBTable
	TIBQuery
	TIBStoredProc
	TIBDatabase
	TIBTransaction
	TIBUpdateSQL
	TIBDataSet
	TIBSQL
	TIBDatabaseInfo
	TIBSQLMonitor

	*TIBEvents*
	TIBExtract
	TIBCustomDataSet

Though they are similar to BDE components in name, the IBX components are somewhat different. For each component with a BDE counterpart, the sections below give a discussion of these differences.

There is no simple migration from BDE to IBX applications. Generally, you must replace BDE components with the comparable IBX components, and then recompile your applications. However, the speed you gain, along with the access you get to the powerful InterBase features make migration well worth your time.

IBDatabase

Use a TIBDatabase component to establish connections to databases, which can involve one or more concurrent transactions. Unlike BDE, IBX has a separate transaction component, which allows you to separate transactions and database connections.

To set up a database connection:

1. Drop an IBDatabase component onto a form or data module.
2. Fill out the DatabaseName property. For a local connection, this is the drive, path, and filename of the database file. Set the Connected property to true.
3. Enter a valid username and password and click OK to establish the database connection.

   Warning: Tip: You can store the username and password in the IBDatabase component's Params property by setting the LoginPrompt property to false after logging in. For example, after logging in as the system administrator and setting the LoginPrompt property to false, you may see the following when editing the Params property:

```
user_name=sysdba
 password=masterkey
```

IBTransaction

Unlike the Borland Database Engine, IBX controls transactions with a separate component, TIBTransaction. This powerful feature allows you to separate transactions and database connections, so you can take advantage of the InterBase two-phase commit functionality (transactions that span multiple connections) and multiple concurrent transactions using the same connection.

Use an IBTransaction component to handle transaction contexts, which might involve one or more database connections. In most cases, a simple one database/one transaction model will do.

To set up a transaction:

1. Set up an IBDatabase connection as described above.
2. Drop an IBTransaction component onto the form or data module
3. Set the DefaultDatabase property to the name of your IBDatabase component.
4. Set the Active property to true to start the transaction.

IBX dataset components

There are a variety of dataset components from which to choose with IBX, each having their own characteristics and task suitability:

IBTable

Use an TIBTable component to set up a live dataset on a table or view without having to enter any SQL statements.

IBTable components are easy to configure:

1. Add an IBTable component to your form or data module.
2. Specify the associated database and transaction components.
3. Specify the name of the relation from the TableName drop-down list.
4. Set the Active property to true.

IBQuery

Use an TIBQuery component to execute any InterBase DSQL statement, restrict your result set to only particular columns and rows, use aggregate functions, and join multiple tables.

IBQuery components provide a read-only dataset, and adapt well to the InterBase client/server environment. To set up an IBQuery component:

1. Set up an IBDatabase connection as described above.
2. Set up an IBTransaction connection as described above.
3. Add an IBQuery component to your form or data module.
4. Specify the associated database and transaction components.
5. Enter a valid SQL statement for the IBQuery's SQL property in the String list editor.
6. Set the Active property to true

IBDataSet

Use an TIBDataSet component to execute any InterBase DSQL statement, restrict your result set to only particular columns and rows, use aggregate functions, and join multiple tables. IBDataSet components are similar to IBQuery components, except that they support live datasets without the need of an IBUpdateSQL component.

The following is an example that provides a live dataset for the COUNTRY table in employee.gdb:

1. Set up an IBDatabase connection as described above.
2. Specify the associated database and transaction components.
3. Add an IBDataSet component to your form or data module.
4. Enter SQL statements for the following properties:

SelectSQL	SELECT Country, Currency FROM Country
RefreshSQL	SELECT Country, Currency FROM Country WHERE Country = :Country
ModifySQL	UPDATE Country SET Country = :Country, Currency = :Currency WHERE Country = :Old_Country
DeleteSQL	DELETE FROM Country WHERE Country = :Old_Country
InsertSQL	INSERT INTO Country (Country, Currency) VALUES (:Country, :Currency)

1. Set the Active property to true.
2.

   Note: Note: Parameters and fields passed to functions are case-sensitive in dialect 3. For example,

```
FieldByName(EmpNo)
```

would return nothing in dialect 3 if the field was 'EMPNO'.

IBStoredProc

Use TIBStoredProc for InterBase executable procedures: procedures that return, at most, one row of information. For stored procedures that return more than one row of data, or "Select" procedures, use either IBQuery or IBDataSet components.

IBSQL

Use an TIBSQL component for data operations that need to be fast and lightweight. Operations such as data definition and pumping data from one database to another are suitable for IBSQL components.

In the following example, an IBSQL component is used to return the next value from a generator:

1. Set up an IBDatabase connection as described above.
2. Put an IBSQL component on the form or data module and set its Database property to the name of the database.
3. Add an SQL statement to the SQL property string list editor, for example:

```
SELECT GEN_ID(MyGenerator, 1) FROM RDB$DATABASE
```

IBUpdateSQL

Use an TIBUpdateSQL component to update read-only datasets. You can update IBQuery output with an IBUpdateSQL component:

1. Set up an IBQuery component as described above.
2. Add an IBUpdateSQL component to your form or data module.
3. Enter SQL statements for the following properties: DeleteSQL, InsertSQL, ModifySQL, and RefreshSQL.
4. Set the IBQuery component's UpdateObject property to the name of the IBUpdateSQL component.
5. Set the IBQuery component's Active property to true.

IBSQLMonitor

Use an TIBSQLMonitor component to develop diagnostic tools to monitor the communications between your application and the InterBase server. When the TraceFlags properties of an IBDatabase component are turned on, active TIBSQLMonitor components can keep track of the connection's activity and send the output to a file or control.

A good example would be to create a separate application that has an TIBSQLMonitor component and a Memo control. Run this secondary application, and on the primary application, activate the TraceFlags of the IBDatabase component. Interact with the primary application, and watch the second's memo control fill with data.

IBDatabaseInfo

Use an TIBDatabaseInfo component to retrieve information about a particular database, such as the sweep interval, ODS version, and the user names of those currently attached to this database.

For example, to set up an IBDatabaseInfo component that displays the users currently connected to the database:

1. Set up an IBDatabase connection as described above.
2. Put an IBDatabaseInfo component on the form or data module and set its Database property to the name of the database.
3. Put a Memo component on the form.
4. Put a Timer component on the form and set its interval.
5. Double click on the Timer's OnTimer event field and enter code similar to the following:

```
Memo1.Text := IBDatabaseInfo.UserNames.Text;   //  Delphi example
Memo1->Text = IBDatabaseInfo->UserNames->Text; //  C++ example
```

IBEvents

Use an *IBEvents* component to register interest in, and asynchronously handle, events posted by an InterBase server.

To set up an IBEvents component:

1. Set up an IBDatabase connection as described above.
2. Put an IBEvents component on the form or data module and set its Database property to the name of the database.
3. Enter events in the Events property string list editor, for example:

```
IBEvents.Events.Add('EVENT_NAME');   //  Delphi example
    IBEvents->Events->Add("EVENT_NAME"); //  C++ Example
```

1. 4. Set the Registered property to true.

2. **InterBase Administration Components**

If you have InterBase installed, you can use the InterBase Administration components, which allow you to use access the powerful InterBase Services API calls.

The components are located on the InterBase Admin tab of the IDE and include:

	TIBConfigService
	TIBBackupService
	TIBRestoreService
	TIBValidationService
	TIBStatisticalService
	TIBLogService
	TIBSecurityService
	TIBLicensingService
	TIBServerProperties
	TIBInstall
	TIBUnInstall

Note: You must install InterBase to use these features.

IBConfigService

Use an TIBConfigService object to configure database parameters, including page buffers, async mode, reserve space, and sweep interval.

IBBackupService

Use an TIBBackupService object to back up your database. With IBBackupService, you can set such parameters as the blocking factor, backup file name, and database backup options.

IBRestoreService

Use an TIBRestoreService object to restore your database. With IBRestoreService, you can set such options as page buffers, page size, and database restore options.

IBValidationService

Use an TIBValidationService object to validate your database and reconcile your database transactions. With the IBValidationService, you can set the default transaction action, return limbo transaction information, and set other database validation options.

IBStatisticalService

Use an TIBStatisticalService object to view database statistics, such as data pages, database log, header pages, index pages, and system relations.

IBLogService

Use an TIBLogService object to create a log file.

IBSecurityService

Use an TIBSecurityService object to manage user access to the InterBase server. With the IBSecurityService, you can create, delete, and modify user accounts, display all users, and set up work groups using SQL roles.

IBLicensingService

Use an TIBLicensingService component to add or remove InterBase software activation certificates.

IBServerProperties

Use an TIBServerProperties component to return database server information, including configuration parameters, and version and license information.

IBInstall

Use an TIBInstall component to set up an InterBase installation component, including the installation source and destination directories, and the components to be installed.

IBUnInstall

Use an TIBUnInstall component to set up an uninstall component.

1.2.5 dbExpress 4 Feature Overview

dbExpress's top level framework and metadata support has been rewritten in Delphi for RAD Studio 2007. It has new, richer metadata support.

The DbxClient driver remotes the dbExpress 4 framework interface over a network based transport.

This document discusses the following features:

- dbExpress Framework
- dbExpress Metadata Improvements
- DBXClient Driver
- DBXDynalink Driver
- DBTest

dbExpress Framework

VCL

The dbExpress VCL component's implementation has changed with minimal change to the API. Most applications are not affected by changes to the dbExpress VCL. However, there are some new methods, properties, events, constants, and enums.

dbExpress Metadata Improvements

New metadata providers for 9 different database backends are written completely in Delphi. Full source code to all metadata providers is included in the product.

Metadata read and write capabilities

Each provider is composed of a metadata reader and metadata writer implementation contained inside the dbExpress driver packages. The separate metadata readers and writers that were in the DbxReadOnlyMetaData and DbxMetaData packages no longer exist.

Provider based approach

The metadata providers are detached from the driver, so that one metadata provider can be used for multiple driver implementations as long as the database backend is the same. Data Explorer also takes advantage of metadata providers to provide metadata support for other database drivers.

The provider is not bound to a driver, but to a database back end. There is a new property called

TDBXPropertyNames.MetaDataPackageLoader in the dbxdrivers.ini files that can be set to a TDBXCommandFactory object. This command factory implementation creates a TDBXCommand that can execute metadata commands. This approach allows multiple driver implementations for a specific database backend to use the same metadata provider. Data Explorer also takes advantage of this architecture to provide dbExpress 4 structured metadata for drivers from other vendors. The decoupling of driver and metadata provider also benefits "thin" driver implementations. If metadata commands can be serviced on a server, there is no need to have the metadata provider logic on the client.

Provider source directory

All database source is now contained in the following installation location:

```
C:\Program FIles \ CodeGear\ RAD Studio \ 6.0 \ source \ database
```

Reading metadata

The unit DBXMetaDataNames has been provided to read metadata. The dbExpress class TDBXMetaDataCommands provides a set of constants to read various types of metadata. Set the TDBXCommand.CommandType property to TDBXCommandTypes.DBXMetadata and set TDBXCommand.Text to one of the constants in TDBXMetaDataCommands to acquire the designated metadata. TDBXCommand.ExecuteQuery returns a TDBXReader to access the metadata. The new classes in DBXMetaDataNames describe and provide access to this metadata's columns.

Writing metadata

Support for creating SQL dialect sensitive CREATE, ALTER, and DROP statements is provided in Data Explorer. dbExpress also exposes a DbxMetaDataProvider class that surfaces this capability for applications. This slightly increases the size of application, since the metadata writers must be included. The ability to generically create tables is useful for many applications. The interface allows you to describe what a table and its columns look like and pass this description to the TdbxMetaDataProvider.CreateTable method. Here is a simple example that shows how to create a table with an int32 column named "C1", a decimal with a precision of 10 and scale of 2 named "C2", and a character based column with a precision of 32 named "C3".

```
var MetaDataTable: TDBXMetaDataTable; DataGenerator: TDbxDataGenerator; Command:
    TDBXCommand; Row: Integer; begin MetaDataTable := TDBXMetaDataTable.Create;
    MetaDataTable.TableName := 'QCXXX_TABLE';
MetaDataTable.AddColumn(TDBXInt32Column.Create('C1'));
    MetaDataTable.AddColumn(TDBXDecimalColumn.Create('C2', 10, 2));
    MetaDataTable.AddColumn(TDBXUnicodeCharColumn.Create('C3', 32));
    MetaDataProvider.CreateTable(MetaDataTable); end
```

Deployment

For information about deploying database applications, see dapdeployingdatabaseapplications.xml (see page 1948).

Compatibility

The VCL components in the SqlExpr unit still work with drivers that provide the more limited dbExpress 3 metadata. However, Data Explorer only works with dbExpress 4 metadata.

Note that Delphi includes metadata for 9 different database backends. Thus any dbExpress driver implementation for the 9 backends supported can reuse the metadata provider with their driver implementation.

DBXClient Driver

DBXClient is a thin dbExpress 4 driver that remotes the dbExpress 4 framework interface over a pluggable network based transport. In this release, a TCP/IP transport is supported. The driver uses a JSON/RPC (Java Script Object Notation) based streaming protocol.

The DBXClient is implemented in 100% Object Pascal. Its source code is included in the product.

Connectivity

DBXClient can connect to Blackfish SQL and DataSnap. Blackfish SQL is a Delphi version of JBuilder's JDataStore. DataSnap provides a middle-tier application server that contains and manages remote data modules. DataSnap has been enhanced to provide a very general connection mechanism between components in different tiers.

To use the DBXClient driver with Blackfish SQL or DataSnap, add the **DBXClient** unit to the **uses** clause.

Deployment

DBXClient needs no database client library installed when you deploy your application. DBXClient is 100% Delphi and can be directly linked into your application as a single .exe file.

For further information about deploying database applications, see dapdeployingdatabaseapplications.xml (see page 1948).

DBTest

This is a collection of classes that extend the capabilities of Dunit to facilitate database testing. The qcreport and cts sample Dunit tests provide good examples of how to make use of DBTest. TestCaseExtension contains non-database related extensions to Dunit and the DBXTest unit contains database related extensions.

Command line properties

New units have been added to the DbxDynalinkDriver package for all 8 of Dynalink drivers:

Test selection

-s:<TestName> command line can be used to execute just a single method in a Dunit test case. This is useful for debugging a single bug. See the **TestCaseExtension** unit.

Convenience methods

There are several methods for creating default connection and metadata provider. See the DBXTest unit.

Data generator

There is a simple, extensible data generator. See the DBXDataGenerator unit.

See Also

dbExpress Framework (see page 26)

TDBXMetaDataCommands

1.2.6 Blackfish SQL Overview

The design and implementation of Blackfish SQL emphasizes database performance, scalability, ease of use, and a strong adherence to industry standards. Blackfish SQL capabilities include the following:

- Industry standards compliance
- Entry level SQL-92
- Unicode storage of character data
- Unicode-based collation key support for sorting and indexing
- dbExpress 4 drivers for win32 Delphi and C++
- ADO.NET 2.0 providers for .NET

- JDBC for Java
- JavaBean data access components for Java
- XA/JTA Distributed transactions for Java
- High performance and scalability for demanding online transaction processing (OLTP) and decision support system (DSS) applications
- Delphi, C#, and VB.NET stored procedures and triggers for Windows
- Java-stored procedures and triggers
- Zero-administration, single assembly or single-jar deployment
- Database incremental backup and failover

Blackfish SQL DataStore

Blackfish SQL is the name of the product, its tools, and of the file format. Within this product, there is a datastore package that includes a DataStore class, as well as several additional classes that have DataStore as part of their names.

Blackfish SQL Compatibility

Blackfish SQL for Windows and Blackfish SQL for Java are highly compatible with one another. The database file format is binary-compatible between Blackfish SQL for Windows and Blackfish SQL for Java. In addition, database clients and servers are interchangeable. Windows clients can connect to Java servers and Java clients can connect to Windows servers.

Because the Blackfish SQL for Windows implementation is more recent, some Blackfish SQL for Java features are not yet supported. The following features are not supported:

- ISQL SQL Command Line Interpreter
- High Availability features, including incremental backup and failover
- Graphical tooling for administrative capabilities
- Access to file and object streams
- Tracking and resolving of row-level insert, update and delete operations
- Access to the Blackfish SQL File System directory

Blackfish SQL Connectivity

This section provides an overview of the connection drivers provided for Blackfish SQL for Windows and Blackfish SQL for Java, respectively. For instructions on using the drivers to connect to a Blackfish SQL database, see the Blackfish SQL Developer's Guide, Establishing Connections section

Blackfish SQL for Windows Connectivity

Blackfish SQL for Windows provides the following connection drivers:

- **DBXClient:** This is a win32 dbExpress 4 database driver that enables win32 Delphi and C++ applications to connect to a remote Blackfish SQL for Windows or Blackfish SQL for Java server.
- **Local ADO.NET 2.0 Provider:** This is a 100% managed code driver that enables .NET applications to connect to a local Blackfish SQL for Windows server. The local ADO.NET driver executes in the same process as the BlackFishSQL database kernel, for greater performance.
- **Remote ADO.NET 2.0 Provider:** This is a 100% managed code driver that enables .NET applications to acquire a remote connection to either a Blackfish SQL for Windows or Blackfish SQL for Java server.

Blackfish SQL for Java Connectivity

Blackfish SQL for Java provides the following JDBC connection drivers:

- **Local JDBC driver:** This is a 100% managed code driver that enables Java applications to connect to a local Blackfish SQL

for Java server. The local JDBC driver executes in the same process as the BlackFishSQL database kernel, for greater performance.

- **Remote JDBC driver:** This is a 100% managed code driver that enables Java applications to acquire a remote connection to either a Blackfish SQL for Windows or Blackfish SQL for Java server.

See Also

Blackfish SQL Developer's Guide: Preface

1

1.2.7 dbExpress Framework

The dbExpress framework (DBX framework) is a set of abstract classes provided in the unit DBXCommon. Applications can interface with the framework in several ways: using the framework directly for both native and managed applications, and using the dbExpress VCL components that are layered on top of the framework for both native and managed applications.

Although many applications interface with dbExpress drivers via the dbExpress VCL components, the DBX framework offers a convenient, lighter weight option to communicate with a database driver. You can also create a database driver for dbExpress by extending the frameworks's DBXCommon abstract base classes. The DBX framework provides most commonly needed database driver functionality for a "set" oriented database application, yet provides a simple interface.

Here are some of the key features of the DBX framework:

- The driver framework is written entirely in Delphi and allows drivers to be written in Delphi.
- It uses strongly typed data access instead of pointers. For instance, it uses String types rather than pointers to strings.
- The driver framework is single sourced. You can compile the source with the native DCC32 compiler.
- The framework has only Abstract base classes that are used for drivers, connections, commands, readers, and so on.
- The framework uses exception based error handling rather than returning error codes.

Capabilities

There are two categories of drivers that extend the classes in DBXCommon: DBXDynaLink and DBXDirect. These drivers differ from each other in the way they are loaded and the capabilities they provide to an application. These are described in greater detail later.

You can also extend the DBX framework to write *delegation drivers*, which provide an extra layer between the application and the actual driver. Delegate drivers are useful for connection pooling, driver profiling, tracing, and auditing. Another possible application of driver delegation is to create a thread safe driver delegate. Such a delegate could provide thread synchronized access to all public methods.

Absolute thread safety is left to applications using dbExpress. However, some thread safety issues are best handled by the dbExpress framework. dbExpress thread safe operations include loading and unloading drivers, and connection creation. As mentioned earlier, a delegate driver can be created to make the entire public interface of dbExpress thread safe if needed.

A dbExpress 4 driver can statically or dynamically link drivers built as Delphi packages. The easiest way to link a driver package is to just include it in the "uses" clause. The driver loader also loads packages specified in a config or ini file using the LoadPackage method. This allows dynamic loading of drivers that are never specified in a uses clause of any of the application's units. Note that the LoadPackage approach can only be employed for applications built to use packages.

dbExpress 4 driver writers should examine the initialization sections of the DBXDynalink and DBXTrace units in the source code provided with dbExpress. These sections register themselves with a singleton unit called the ClassRegistry. The ClassRegistry is used by the dbExpress 4 driver loader to instantiate driver loader classes by name (a String). The ClassRegistry is a simple, lightweight mechanism for registering and instantiating a class by name.

DBXDynalink Drivers

DBXDynalink is used for existing dbExpress 3 drivers as well as new drivers. It is compiled as a native Delphi package. DBXDynalink loads native dbExpress drivers that implement a more primitive "native" interface called DBXExports. The DBXExports interface is a small collection of "flat" export methods. DBXExports's source is included with dbExpress. DBXExports provides a more strongly typed API than the dbExpress 3's COM based interface. This allows methods to be added in future product generations without breaking compatibility with older implementations of the DBXExports interface.

DBXAdapter is a dbExpress 4 compliant driver that adapts the DBXExports interface to the older dbExpress 3 COM interface. Newer native drivers can be written by implementing DBXExports directly.

Because the DBXExports interface is designed to be implemented using any native language (Delphi or C++), it uses more primitive, non-exception based error handling. DBXDynalink maps error codes to a DBXCommon exception.

The DBXDynalink unit contains a dbExpress 4 driver. This driver delegates to non-Delphi drivers that implement the DBXDynalinkExport flat export interface. DBXTrace is a delegate driver used for tracing. The dbExpress VCL uses DBXCommon, DBXDynalink and DbxTrace as "default" drivers. However, this can be changed for statically linked applications without modifying dbExpress VCL source code (SQLExpr.pas). SQLExpr.pas uses the unit DBXDefaultDrivers. The DBXDefaultDrivers unit only contains a uses clause. The DBXDefaultDrivers uses clause contains DBXCommon, DBXDynalink, and DBXTrace. DBXCommon must always be used. However, a statically linked application could remove DBXTrace and replace DBXDynalink with a different driver.

DBXDirect Drivers

A DBXDirect driver is any driver that is implemented by extending the DBXCommon abstract base classes. These classes are written in Delphi for native implementations.

Strictly speaking, all DBX framework drivers are a form of DBXDirect driver. However DBXDynalink and DBXRemote provide a more "indirect" linkage to driver implementations.

See Also

dbExpress Framework Compatibility (see page 27)

Deploying dbExpress Database Applications (see page 1942)

1.2.8 dbExpress Framework Compatibility

Some dbExpress software developed prior to the dbExpress driver framework (DBX driver framework) has been modified to work with the DBX driver framework. As a result of these changes, some compatibility issues arise.

General

dbExpress 2.5 drivers *cannot* be used with the DBX framework.

The dbExpress framework does not provide 100% compatibility with dbExpress 3.

dbExpress 3 drivers can be used with the DBX framework. The DBX framework driver loader automatically detects dbExpress 3 drivers and uses the DBXAdapter driver (dbxadapter30.dll) to make a dbExpress 3 driver look like a dbExpress 4 driver.

Here is a list of known compatibility issues:

- Static driver linkage. You cannot statically link dbExpress drivers into an executable.
- SqlExpr.TSQLConnection provided protected access to the Connection member that was of type TISQLConnection only in the native version of SqlExpr.pas. This was omitted from the managed version due to the complexity of how PInvoke was used in the managed version of the dbExpress VCL. SqlExpr.TSQLConnection now provides protected access to a TDBXConnection instead. This protected connection is accessible to both native and managed applications.

- The event for trace monitoring is slightly different because it is based on the DBX driver framework.
- The DBXadapter driver can adapt dbExpress 3 drivers to dbExpress 4, but not dbExpress 2.5 drivers.

VCL issues

Most applications using dbExpress VCL components should work without modification. However, there are some localized changes to VCL components due to VCL now interfacing to the more object oriented DBX driver framework instead of the C-like COM-based dbExpress 3 driver interface.

In addition, the API has changed slightly for two of the VCL components: TSQLConnection and TSQLDataSet. Some data structures have also changed. A summary of the API changes follows.

Note: Because of API changes, you must recompile SqlExpr.pas, which is provided with the product. The DBXpress unit has been deprecated.

- TSQLConnection. The Commit method has been deprecated in favor of the new CommitFreeAndNil method. The Rollback method has been deprecated in favor of the new RollbackFreeAndNil and RollbackIncompleteFreeAndNil methods. The SetTraceCallbackEvent method has been replaced by SetTraceEvent. The StartTransaction method has been deprecated in favor of the new BeginTransaction method. The MetaData property contains an instance of the new class TDBXDatabaseMetaData instead of TISQLMetaData. The SQLConnection property has been replaced by DBXConnection, which contains an instance of the new class TDBXConnection. The TraceCallbackEventproperty now contains a TDBXTraceEvent.
- TSQLDataSet. A new property DbxCommandType has been added, which contains one of the constant strings in the TDBXCommandTypes class.
- Data structures. TTransactionItem has been deprecated, replaced by the new TDBXTransaction class. TSQLDriverOption, TSQLConnectionOption, TSQLCommandOption, TSQLCursorOption, TSQLMetaDataOption, and TSQLObjectType are obsolete. TSTMTParamType has been replaced by the TDBXParameterDirections class. TSQLTraceFlag has been replaced by TDBXTraceFlags. SQLTRACEDesc is replaced by TDBXTraceInfo.

See Also

dbExpress Framework (see page 26)

Deploying dbExpress Database Applications (see page 1942)

1.3 Developing Interoperable Applications

RAD Studio provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms.

Topics

Name	Description
Developing COM Applications (see page 29)	Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients and servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms. This topic covers: • COM Technologies Overview • COM Interfaces • COM Servers • COM Clients

1.3.1 Developing COM Applications

Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients and servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms.

This topic covers:

- COM Technologies Overview
- COM Interfaces
- COM Servers
- COM Clients

COM Technologies Overview

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The most important aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM).

COM is both a specification and an implementation. The COM specification defines how objects are created and how they communicate with each other. According to this specification, COM objects can be written in different languages, run in different

process spaces and on different platforms. As long as the objects conform to the written specification, they can communicate. This allows you to integrate legacy code as a component with new components implemented in object-oriented languages.

The COM implementation is built into the Win32 subsystem, which provides a number of core services that support the specification. The COM library contains a set of standard interfaces that define the core functionality of a COM object, and a small set of API functions for creating and managing COM objects.

When you use Delphi wizards and VCL objects in your application, you are using Delphi's implementation of the COM specification. In addition, Delphi provides some wrappers for COM services for those features that it does not implement directly (such as Active Documents). You can find these wrappers defined in the `ComObj` unit and the API definitions in the `AxCtrls` unit.

Note: Delphi's interfaces and language follow the COM specification. Delphi implements objects conforming to the COM spec using a set of classes called the Delphi ActiveX framework (DAX). These classes are found in the `AxCtrls`, `OleCtrls`, and `OleServer` units. In addition, the Delphi interface to the COM API is in `ActiveX.pas` and `ComSvcs.pas`.

COM Interfaces

COM clients communicate with objects through COM interfaces. Interfaces are groups of logically or semantically related routines which provide communication between a provider of a service (server object) and its clients.

For example, every COM object must implement the basic interface, `IUnknown`. Through a routine called `QueryInterface` in `IUnknown`, clients can request other interfaces implemented by the server.

Objects can have multiple interfaces, where each interface implements a feature. An interface provides a way to tell the client what service it provides, without providing implementation details of how or where the object provides this service.

Key aspects of COM interfaces are as follows:

- Once published, interfaces do not change. You can rely on an interface to provide a specific set of functions. Additional functionality is provided by additional interfaces.
- By convention, COM interface identifiers begin with a capital I and a symbolic name that defines the interface, such as `IMalloc` or `IPersist`.
- Interfaces are guaranteed to have a unique identification, called a Globally Unique Identifier (GUID), which is a 128-bit randomly generated number. Interface GUIDs are called Interface Identifiers (IIDs). This eliminates naming conflicts between different versions of a product or different products.
- Interfaces are language independent. You can use any language to implement a COM interface as long as the language supports a structure of pointers, and can call a function through a pointer, either explicitly or implicitly.
- Interfaces are not objects themselves, they provide a way to access an object. Therefore, clients do not access data directly, they access data through an interface pointer. Windows 2000 adds another layer of indirection, known as an interceptor, through which it provides COM+ features such as just-in-time activation and object pooling.
- Interfaces are always inherited from the base interface, `IUnknown`.
- Interfaces can be redirected by COM through proxies to enable interface method calls to call between threads, processes, and networked machines, all without the client or server objects ever being aware of the redirection.

The IUnknown interface

All COM objects must support the fundamental interface, called `IUnknown`, a typedef to the base interface type `IInterface`. `IUnknown` contains the following routines:

- `QueryInterface`: Provides pointers to other interfaces that the object supports.
- `AddRef` and `Release`: Simple reference counting methods that keep track of the object's lifetime so that an object can delete itself when the client no longer needs its service.

Clients obtain pointers to other interfaces through the `IUnknown` method, `QueryInterface`. `QueryInterface` knows about every interface in the server object and can give a client a pointer to the requested interface. When receiving a pointer to an

interface, the client is assured that it can call any method of the interface.

Objects track their own lifetime through the `IUnknown` methods, `AddRef` and `Release`, which are simple reference counting methods. As long as the reference count of an object is nonzero, the object remains in memory. Once the reference count reaches zero, the interface implementation can safely dispose of the underlying object.

COM Interface Pointers

An interface pointer is a pointer to an object instance that points, in turn, to the implementation of each method in the interface. The implementation is accessed through an array of pointers to these methods, which is called a vtable. Vtables are similar to the mechanism used to support virtual functions in Delphi. Because of this similarity, the compiler can resolve calls to methods on the interface the same way it resolves calls to methods on Delphi classes.

The vtable is shared among all instances of an object class, so for each object instance, the object code allocates a second structure that contains its private data. The client's interface pointer, then, is a pointer to the pointer to the vtable.

In Windows 2000 and subsequent versions of Windows, when an object is running under COM+, another level of indirection is provided between the interface pointer and the vtable pointer. The interface pointer available to the client points at an interceptor, which in turn points at the vtable. This allows COM+ to provide such services as just-in-time activation, where the server can be deactivated and reactivated dynamically in a way that is opaque to the client. To achieve this, COM+ guarantees that the interceptor behaves as if it were an ordinary vtable pointer.

COM Servers

A COM server is an application or a library that provides services to a client application or library. A COM server consists of one or more COM objects, where a COM object is a set of properties and methods.

Clients do not know how a COM object performs its service; the object's implementation remains hidden. An object makes its services available through its interfaces as described previously.

In addition, clients do not need to know where a COM object resides. COM provides transparent access regardless of the object's location.

When a client requests a service from a COM object, the client passes a class identifier (CLSID) to COM. A CLSID is simply a GUID that identifies a COM object. COM uses this CLSID, which is registered in the system registry, to locate the appropriate server implementation. Once the server is located, COM brings the code into memory, and has the server create an object instance for the client. This process is handled indirectly, through a special object called a class factory (based on interfaces) that creates instances of objects on demand.

As a minimum, a COM server must perform the following:

- Register entries in the system registry that associate the server module with the class identifier (CLSID).
- Implement a class factory object, which creates another object of a particular CLSID.
- Expose the class factory to COM.
- Provide an unloading mechanism through which a server that is not servicing clients can be removed from memory.

COM Clients

COM clients are applications that make use of a COM object implemented by another application or library. The most common types are Automation controllers, which control an Automation server and ActiveX containers, which host an ActiveX control.

There are two types of COM clients, controllers and containers. Controllers launch the server and interact with it through its interface. They request services from the COM object or drive it as a separate process. Containers host visual controls or objects that appear in the container's user interface. They use predefined interfaces to negotiate display issues with server objects. It is impossible to have a container relationship over DCOM; for example, visual controls that appear in the container's user interface must be located locally. This is because the controls are expected to paint themselves, which requires that they have access to local GDI resources.

The task of writing these two types of COM client is remarkably similar: The client application obtains an interface for the server

object and uses its properties and methods. Delphi makes it easier for you to develop COM clients by letting you import a type library or ActiveX control into a component wrapper so that server objects look like other VCL components. Delphi lets you wrap the server `CoClass` in a component on the client, which you can even install on the Component palette. Samples of such component wrappers appear on two pages of the Component palette, sample ActiveX wrappers appear on the ActiveX page, and sample Automation objects appear on the Servers page.

Even if you do not choose to wrap a server object in a component wrapper and install it on the Component palette, you must make its interface definition available to your application. To do this, you can import the server's type information.

Clients can always query the interfaces of a COM object to determine what it is capable of providing. All COM objects allow clients to request known interfaces. In addition, if the server supports the `IDispatch` interface, clients can query the server for information about what methods the interface supports. Server objects have no expectations about the client using its objects. Similarly, clients don't need to know how an object provides the services, they simply rely on server objects to provide the services they describe in their interfaces.

COM Extensions

As COM has evolved, it has been extended beyond the basic COM services. COM serves as the basis for other technologies such as Automation, ActiveX controls, Active Documents, and Active Directories. In addition, when working in a large, distributed environment, you can create transactional COM objects. Prior to Windows 2000, these objects were not an architectural part of COM, but ran in the Microsoft Transaction Server (MTS) environment. As of Windows 2000, this support is integrated into COM+. Delphi provides wizards to easily implement applications that use the above technologies in the Delphi environment.

Automation Servers

Automation refers to the ability of an application to control the objects in another application programmatically, such as a macro that can manipulate more than one application at the same time. The server object being manipulated is called the Automation object, and the client of the Automation object is referred to as an Automation controller. Automation can be used on in-process, local, and remote servers.

Automation is defined by two major points:

- The Automation object defines a set of properties and commands, and describes their capabilities through type descriptions. In order to do this, it must have a way to provide information about its interfaces, the interface methods, and the arguments to those methods. Typically, this information is available in a type library. The Automation server can also generate type information dynamically when queried via its `IDispatch` interface.
- Automation objects make their methods accessible so that other applications can use them. For this, they implement the `IDispatch` interface. Through this interface an object can expose all of its methods and properties. Through the primary method of this interface, the object's methods can be invoked, once having been identified through type information.

Developers often use Automation to create and use non-visual OLE objects that run in any process space, because the Automation `IDispatch` interface automates the marshaling process. Automation does, however, restrict the types that you can use.

Active X Controls

Delphi wizards allow you to easily create ActiveX controls. ActiveX is a technology that allows COM components, especially controls, to be more compact and efficient. This is especially necessary for controls that are intended for Intranet applications, which need to be downloaded by a client before they are used.

ActiveX controls are visual controls that run only as in-process servers, and can be plugged into an ActiveX control container application. They are not complete applications in themselves, but can be thought of as already written OLE controls that are reusable in various applications. ActiveX controls have a visible user interface, and rely on predefined interfaces to negotiate I/O and display issues with their host containers.

ActiveX controls make use of Automation to expose their properties, methods, and events. Features of ActiveX controls include the ability to fire events, bind to data sources, and support licensing.

One use of ActiveX controls is on a Web site as interactive objects in a Web page. As such, ActiveX is a standard that targets

interactive content for the World Wide Web, including the use of ActiveX Documents used for viewing non-HTML documents through a Web browser. For more information about ActiveX technology, see the Microsoft ActiveX Web site.

Active Documents

Active Documents (previously referred to as OLE documents) are a set of COM services that support linking and embedding, drag-and-drop, and visual editing. Active Documents can seamlessly incorporate data or objects of different formats, such as sound clips, spreadsheets, text, and bitmaps.

Unlike ActiveX controls, Active Documents are not limited to in-process servers; they can be used in cross-process applications.

Unlike Automation objects, which are almost never visual, Active Document objects can be visually active in another application. Thus, Active Document objects are associated with two types of data: presentation data, used for visually displaying the object on a display or output device, and native data, used to edit an object.

Active Document objects can be document containers or document servers. While Delphi does not provide an automatic wizard for creating Active Documents, you can use the VCL class, TOleContainer, to support linking and embedding of existing Active Documents.

You can also use TOleContainer as a basis for an Active Document container. To create objects for Active Document servers, use the COM object wizard and add the appropriate interfaces, depending on the services the object needs to support. For more information about creating and using Active Document servers, see the Microsoft ActiveX Web site.

Note: While the specification for Active Documents has built-in support for marshaling in cross-process applications, Active Documents do not run on remote servers because they use types that are specific to a system on a given machine such as window handles, menu handles, and so on.

Transactional Objects

Delphi uses the term "transactional objects" to refer to objects that take advantage of the transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) (for versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later). These objects are designed to work in a large, distributed environment.

The transaction services provide robustness so that activities are always either completed or rolled back. The server never partially completes an activity. The security services allow you to expose different levels of support to different classes of clients. The resource management allows an object to handle more clients by pooling resources or keeping objects active only when they are in use. To enable the system to provide these services, the object must implement the `IObjectControl` interface. To access the services, transactional objects use an interface called `IObjectContext`, which is created for them by MTS or COM+.

Under MTS, the server object must be built into a DLL library, which is then installed in the MTS runtime environment. That is, the server object is an in-process server that runs in the MTS runtime process space. Under COM+, this restriction does not apply because all COM calls are routed through an interceptor. To clients, the difference between MTS and COM+ is transparent.

MTS or COM+ servers group transactional objects that run in the same process space. Under MTS, this group is called an MTS package, while under COM+ it is called a COM+ application. A single machine can be running several different MTS packages (or COM+ applications), where each one is running in a separate process space.

To clients, the transactional object may appear like any other COM server object. The client does not need know about transactions, security, or just-in-time activation unless it is initiating a transaction itself.

Both MTS and COM+ provide a separate tool for administering transactional objects. This tool lets you configure objects into packages or COM+ applications, view the packages or COM+ applications installed on a computer, view or change the attributes of the included objects, monitor and manage transactions, make objects available to clients, and so on. Under MTS, this tool is the MTS Explorer. Under COM+ it is the COM+ Component Manager.

Type Libraries

Type libraries provide a way to get more type information about an object than can be determined from an object's interface. The type information contained in type libraries provides needed information about objects and their interfaces, such as what interfaces exist on what objects (given the CLSID), what member functions exist on each interface, and what arguments those functions require.

You can obtain type information either by querying a running instance of an object or by loading and reading type libraries. With this information, you can implement a client which uses a desired object, knowing specifically what member functions you need, and what to pass those member functions.

Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available. All of Delphi's wizards generate a type library automatically, although the COM object wizard makes this optional. You can view or edit this type information by using the Type Library Editor.

See Also

Using COM Wizards (see page 79)

1.4 Developing Reports for Your Win32 Applications

RAD Studio ships with Rave Reports from Nevrona. Using the report components, you can build full-featured reports for your applications. You can create solutions that include reporting capabilities which can be used and customized by your customers. Additionally, the ComponentOne tools that ship with RAD Studio include components for creating and generating reports.

Topics

Name	Description
Using Rave Reports in RAD Studio (see page 35)	The RAD Studio environment supports the integration of report objects in your applications. This integration allows you to create a report using the Rave Reports Designer directly from within the RAD Studio IDE. Your application users can create and display their own reports, or display existing reports.

1.4.1 Using Rave Reports in RAD Studio

The RAD Studio environment supports the integration of report objects in your applications. This integration allows you to create a report using the Rave Reports Designer directly from within the RAD Studio IDE. Your application users can create and display their own reports, or display existing reports.

Creating New Reports in RAD Studio

You can include reports in RAD Studio just as you would other 3rd-party components. The report is stored as a separate Rave Report object. You can reference the report in other applications that need to call or generate that report. When you create a new application, you can include the report object by adding a reference to it in the **Project Manager**. Rave Reports also provide the capability to connect your report object to a datasource, which allows your application to build the report dynamically, based on current database information.

See Also

Adding Rave Reports to RAD Studio (see page 81)

1.5 Developing Applications with VCL Components

The Visual Component Library (VCL) is a set of visual components for the rapid development of Windows applications in the Delphi language.

VCL contains a wide variety of visual, non-visual, and utility classes for tasks such as building Windows applications, web applications, database applications, and console applications.

Topics

Name	Description
VCL Overview (see page 36)	This section introduces: • VCL Architecture • VCL Components • Working With Components
Using TEncoding for Unicode Files (see page 38)	
Components Available Only on Specific OS (see page 39)	Some VCL components are specific to an operating system (such as Vista) and thus will not work on other operating systems. For example, the design time components TFileOpenDialog and TFileSaveDialog are specific to Vista. Some components are designed to work with themes either enabled or disabled. (To enable or disable themes, check or uncheck the Enable runtime themes box on the **Project** ▶ **Options** ▶ **Application** dialog box.) In the following table, note that Windows Server 2003 is treated as Windows XP, and Windows Server 2008 is treated as Vista. ***Components Available Only for Certain Operating Systems***

1.5.1 VCL Overview

This section introduces:

- VCL Architecture
- VCL Components
- Working With Components

VCL Architecture

VCL is an acronym for the Visual Component Library, a set of visual components for rapid development of Windows applications in the Delphi language. VCL contains a wide variety of visual, non-visual, and utility classes for tasks such as Windows application building, web applications, database applications, and console applications. All classes descend from System::TObject. System::TObject introduces methods that implement fundamental behavior like construction, destruction, and message handling.

VCL Components

Components are a subset of the component library that descend from the class TComponent. You can place components on a form or data module and manipulate them at designtime. Using the **Object Inspector**, you can assign property values without writing code. Most components are either visual or nonvisual, depending on whether they are visible at runtime. Some components appear on the **Component Palette**.

Visual Components

Visual components, such as TForm and TSpeedButton, are called controls and descend from TControl. Controls are used in GUI applications, and appear to the user at runtime. TControl provides properties that specify the visual attributes of controls, such as their height and width.

NonVisual Components

Nonvisual components are used for a variety of tasks. For example, if you are writing an application that connects to a database, you can place a TDataSource component on a form to connect a control and a dataset used by the control. This connection is not visible to the user, so TDataSource is nonvisual. At designtime, nonvisual components are represented by an icon. This allows you to manipulate their properties and events just as you would a visual control.

Other VCL Classes

Classes that are not components (that is, classes that descend from System::TObject but not TComponent) are also used for a variety of tasks. Typically, these classes are used for accessing system objects (such as a file or the clipboard) or for transient tasks (such as storing data in a list). You cannot create instances of these classes at designtime, although they are sometimes created by the components that you add in the **Form Designer**.

Working With Components

Many components are provided in the IDE on the **Component Palette**. You select components from the **Component Palette** and place them onto a form or data module. You design the user interface of an application by arranging the visual components such as buttons and list boxes on a form. You can also place nonvisual components, such as data access components, on either a form or a data module. At first, Delphi's components appear to be just like any other classes. But there are differences between components in Delphi and the standard class hierarchies that many programmers work with. Some differences are:

- All Delphi components descend from TComponent.
- Components are most often used as is. They are changed through their properties, rather than serving as base classes to be subclassed to add or change functionality. When a component is inherited, it is usually to add specific code to existing event handling member functions.
- Components can only be allocated on the heap, not on the stack.
- Properties of components contain runtime type information.
- Components can be added to the **Component Palette** in the IDE and manipulated on a form.

Components often achieve a better degree of encapsulation than is usually found in standard classes. For example, consider a dialog box containing a button. In a Windows program developed using VCL components, when a user clicks the button, the system generates a `WM_LBUTTONDOWN` message. The program must catch this message (typically in a switch statement, a message map, or a response table) and send it to a routine that will execute in response to the message. Most Windows messages (VCL applications) are handled by Delphi components. When you want to respond to a message or system event, you only need to provide an event handler.

Using Events

Almost all the code you write is executed, directly or indirectly, in response to events. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event, called an event handler, is a Delphi procedure.

The **Events** page of the **Object Inspector** displays all events defined for a given component. Double-clicking an event in the **Object Inspector** generates a skeleton event handling procedure, which you can fill in with code to respond to that event. Not all components have events defined for them.

Some components have a default event, which is the event the component most commonly needs to handle. For example, the default event for a button is `OnClick`. Double-clicking on a component with a default event in the **Form Designer** will generate a skeleton event handling procedure for the default event.

You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop down menu commands. When a button performs the same action as a menu command, you can write a single event handler and then assign it to the OnClick event for both the button and the menu item by setting the event handler in the **Object Inspector** for both the events you want to respond to.

This is the simplest way to reuse event handlers. However, action lists, and in the VCL, action bands, provide powerful tools for centrally organizing the code that responds to user commands. Action lists can be used in cross-platform applications; action bands cannot.

Setting Component Properties

To set published properties at design time, you can use the **Object Inspector** and, in some cases, property editors. To set properties at runtime, assign their values in your application source code.

When you select a component on a form at design time, the **Object Inspector** displays its published properties and, when appropriate, allows you to edit them.

When more than one component is selected, the **Object Inspector** displays all properties—except `Name`—that are shared by the selected components. If the value for a shared property differs among the selected components, the **Object Inspector** displays either the default value or the value from the first component selected. When you change a shared property, the change applies to all selected components.

Changing code-related properties, such as the name of an event handler, in the **Object Inspector** automatically changes the corresponding source code. In addition, changes to the source code, such as renaming an event handler method in a form class declaration, are immediately reflected in the **Object Inspector**.

See Also

Building a VCL Forms Application (see page 99)

Building a VCL Forms "Hello world" Application (see page 101)

Organizing Actions for Toolbars and Menus (see page 1992)

1.5.2 Using TEncoding for Unicode Files

Reading and Writing in the Old Format

Many Delphi applications will need to continue to interact with other applications or datasources, many of which can only handle data in ANSI or ASCII. For this reason, the defaults for the TStrings methods will write the files ANSI encoded based on the active code page and will read the files based on whether or not the file contains a Byte Order Mark (BOM).

If a BOM is found, it will read the data encoded as the BOM indicates. If no BOM is found, it will read it as ANSI and up-convert based on the current active codepage.

All your files written with pre-RAD Studio 2009 versions of Delphi will still be read in, with the caveat that as long as you read with the active codepage the same as with what was written. Likewise, any file written with RAD Studio 2009 with an ASCII encoding should be readable with the pre-RAD Studio 2009 version.

Any file written with RAD Studio 2009 with any other encoding will generate a BOM and will not be readable with a the pre-RAD Studio 2009 version. At this point, only the most common BOM formats are detected (UTF16 Little-Endian, UTF16 Big-Endian and UTF8).

Using the new encodings

You may want to read/write text data using the TStrings class a loss-less Unicode format, be that Little-Endian UTF16, Big-Endian UTF16, UTF8, UTF7 so on. The TEncoding class is very similar in methods and functionality that you can find in the

System.Text.Encoding class in the .NET Framework.

```
var
  S: TStrings;
begin
  S: TStringList.Create();
  { ... }
  S.SaveToFile('config.txt', TEncoding UTF8);
```

Without the extra parameter, ‘config.txt’ would simply be converted and written out as ANSI encoded based on the current active codepage. You do not need to change the read code since TStrings will automatically detect the encoding based on the BOM and do the right thing.

If you wanted to force the file to read and write using a specific codepage, you can create an instance of TMBCSEncoding and pass in the code page you want to use into the constructor. Then you use that instance to read and write the file, since the specific codepage may not match the user’s active codepage.

The same thing holds for these classes in that the data will be read and written as ANSI data. Since INI files have always been traditionally ANSI (ASCII) encoded, it may not make sense to convert these. It will depend on the needs of your application. If you do wish to change to use a Unicode format, we will offer ways to use the TEncoding classes to accomplish that as well.

In all the above cases, the internal storage will be Unicode and any data manipulation you do with string will continue to function as expected. Conversions will automatically happen when reading and writing the data.

Here is a list of codepage identifiers (MSDN).:

The following list shows the overload methods that accept a TEncoding parameter:

- WriteNode (Outline.TOutlineNode)
- LoadFromFile (Outline.TOutlineNode)
- LoadFromStream (Outline.TOutlineNode)
- SaveToFile (Outline.TOutlineNode)
- SaveToStream (Outline.TOutlineNode)
- LoadFromFile (Classes.TStrings)
- LoadFromStream (Classes.TStrings)
- SaveToFile (Classes.TStrings)
- SaveToStream (Classes.TStrings)
- Create (Classes.TStringStream)
- Create (Classes.TStreamReader)
- Create (Classes.TStreamWriter)

1.5.3 Components Available Only on Specific OS

Some VCL components are specific to an operating system (such as Vista) and thus will not work on other operating systems. For example, the design time components TFileOpenDialog and TFileSaveDialog are specific to Vista. Some components are designed to work with themes either enabled or disabled. (To enable or disable themes, check or uncheck the Enable runtime themes box on the **Project** ▸ **Options** ▸ **Application** dialog box.)

In the following table, note that Windows Server 2003 is treated as Windows XP, and Windows Server 2008 is treated as Vista.

Components Available Only for Certain Operating Systems

Component	Windows 2000	Windows XP	Windows Vista
CheckBox, Checked, and FixedWidth properties for THeaderSection	No	No	Yes
CheckBoxes, NoSizing and OverFlow properties to THeaderControl	No	No	Yes
Group support for TListView	No	Yes	Yes
ImageList support for TButton	No	Yes	Yes
Platform Default Style for ActionMenus/ActionToolBars	Yes, standard style	Yes, XP style	Yes, Vista style
Support of transparent PNGs in TImage	Yes	Yes	Yes
TBalloonHints, Vista style	No	Yes, with themes enabled	Yes
TButton.ElevationRequired	No	No	Yes
TButton.CommandLink	No	No	Yes
TButton.SplitButton	No	No	Yes
TComboBox.TextHint	No	No	Yes
TEdit.TextHint	No	No	Yes
TGlassFrame	No	No	Yes
TImageList.ColorDepth := cd32Bit	No	Yes, with themes enabled	Yes, with themes enabled
TLabel.GlowSize (only on TGlassFrame)	No	No	Yes
TLinkLabel	No (plain text only)	Yes, with themes enabled	Yes, with themes enabled
TListView collapsible groups	No	No	Yes
TProgessBar colors and progress automatically set for state by the engines	No	Yes, with themes enabled	Yes
TProgressBar.SmoothReverse	No	No	Yes
TStaticText support HTML links in the textfield. AllowLInks property	No	Yes, with themes enabled	Yes
TTaskDialog	No	No	Yes

See Also

Building a VCL Forms Application (see page 99)

Building a VCL Forms "Hello world" Application (see page 101)

1.6 Developing Web Applications with WebSnap

This section provides a conceptual background for building WebSnap applications using RAD Studio. WebSnap makes it easier to build Web server applications that deliver complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripting makes development and maintenance easier for teams of developers and Web designers.

Please note that WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

Topics

Name	Description
Win32 Web Applications Overview (see page 41)	This section covers: • Web Application Support • Web Broker Overview • Web Snap Overview • Debugging With the Web Application Debugger For more detailed information on web applications, please see the Win32 Developers Guide in the Reference section of this Help system.

1.6.1 Win32 Web Applications Overview

This section covers:

- Web Application Support
- Web Broker Overview
- Web Snap Overview
- Debugging With the Web Application Debugger

For more detailed information on web applications, please see the Win32 Developers Guide in the Reference section of this Help system.

Win32 Web Application Support

The following types of web applications will be supported in RAD Studio.

- ISAPI
- CGI
- Web Application Debugger

Apache web applications are not supported for this release.

ISAPI

Selecting this type of application sets up your project as a DLL, with the exported methods expected by the Web server. It adds the library header to the project file, and the required entries to the `uses` list and `exports` clause of the project file.

CGI

Selecting this type of application sets up your project as a console application, and adds the required entries to the `uses` clause of the project file.

Web Application Debugger

Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment.

Web Broker Overview

Web Broker components, located on the **Internet** tab of the **Component Palette**, enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can construct HTML or XML documents within your program and transfer them to the client. You can use Web Broker components for cross-platform application development.

Frequently, the content of Web pages is drawn from databases. You can use Internet components to automatically manage connections to databases, allowing a single DLL to handle multiple simultaneous, thread-safe, database connections.

Web Snap Overview

Note: WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

WebSnap augments Web Broker with additional components, wizards, and views, making it easier to build Web server applications that deliver complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripting makes development and maintenance easier for teams of developers and Web designers. WebSnap allows HTML design experts on your team to make a more effective contribution to Web server development and maintenance.

The final product of the WebSnap development process includes a series of scriptable HTML page templates. These pages can be changed using HTML editors that support embedded script tags, like Microsoft FrontPage, or even a text editor. Changes can be made to the templates as needed, even after the application is deployed. There is no need to modify the project source code at all, which saves valuable development time. WebSnap’s multiple module support can be used to divide your application into smaller pieces during the coding phases of your project, so that developers can work more independently.

Debugging With the Web Application Debugger

The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the supported types of Web application and install it with a commercial Web server.

To use the Web Application Debugger, you must first create your Web application as a Web Application Debugger executable. Whether you are using Web Broker or WebSnap, the wizard that creates your Web server application includes this as an option when you first begin the application. This creates a Web server application that is also a COM server. The first time you run your application, it registers your COM server so that the Web Application Debugger can access it. Before you can run the Web Application Debugger, you will need to run `bin\serverinfo.exe` once to register the ServerInfo application.

Launching your application with the Web Application Debugger

Once you have developed your Web server application, you can run and debug it using the Web Application Debugger. You can set breakpoints in it just like any other executable. When you run your application, it displays the console window of the COM server that is your Web server application. Once you start your application and run the Web App Debugger, the ServerInfo page is displayed in your default browser, and you can select your application from a drop-down list. Once you have selected your application, click the **Go** button. This launches your application in the Web Application Debugger, which provides you with details on request and response messages that pass between your application and the Web Application Debugger.

Converting your application to another type of Web server application after debugging

When you have finished debugging your Web server application with the Web Application Debugger, you will need to convert it to another type that can be installed on a commercial Web server.

See Also

Building a WebSnap Application (see page 148)

Building a WebSnap "Hello world" Application (see page 149)

Debugging a WebSnap Application Using the Web Application Debugger (see page 150)

Converting Your Application to Another Type of Web Server Application (see page 2245)

Using IntraWeb (see page 2254)

1.7 Developing Web Services with Win32 Applications

Web Services are self-contained modular applications that can be published and invoked over the Internet. Web Services provide well-defined interfaces that describe the services provided. Unlike Web server applications that generate Web pages for client browsers, Web Services are not designed for direct human interaction. Rather, they are accessed programmatically by client applications. This section gives an overview of web services and web services support.

Topics

Name	Description
Web Services Overview (see page 44)	Web Service applications are server implementations that do not require clients to use a specific platform or programming language. These applications define interfaces in a language-neutral document, and they allow multiple communication mechanisms. Web Services are designed to work using Simple Object Access Protocol (SOAP). SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. SOAP uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol. Web Service applications use a Web Service Definition Language (WSDL) document to publish information on interfaces that are available and how to call them. On... more (see page 44)

1.7.1 Web Services Overview

Web Service applications are server implementations that do not require clients to use a specific platform or programming language. These applications define interfaces in a language-neutral document, and they allow multiple communication mechanisms.

Web Services are designed to work using Simple Object Access Protocol (SOAP). SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. SOAP uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol.

Web Service applications use a Web Service Definition Language (WSDL) document to publish information on interfaces that are available and how to call them. On the server side, your application can publish a WSDL document that describes your Web Service. On the client side, a wizard or command-line utility can import a published WSDL document, providing you with the interface definitions and connection information you need. If you already have a WSDL document that describes the Web service you want to implement, you can generate the server-side code when you import the WSDL document.

See Also

Using Web Services (see page 2291)

1.8 Developing Windows Applications

Windows provides a traditional approach to developing user interfaces, client/server applications, controls, and application logic. This section provides an overview of Windows application development using RAD Studio for Win32 and outlines the steps you would use to build a simple Windows project.

Topics

Name	Description
Windows Overview (see page 45)	The Windows platform provides several ways to help you create and build applications. The most common types of Windows applications are: • GUI Applications • Console Applications • Service Applications • Packages and DLLs

1.8.1 Windows Overview

The Windows platform provides several ways to help you create and build applications. The most common types of Windows applications are:

- GUI Applications
- Console Applications
- Service Applications
- Packages and DLLs

GUI Applications

A graphical user interface (GUI) application is designed using graphical components such as windows, menus, dialog boxes, and other features that make the application easy to use. When you compile a GUI application, an executable file with start-up code is created from your source files. The executable usually provides the basic functionality of your program. Simple programs often consist of only an executable file. You can extend the application by calling DLLs, packages, and other support files from the executable.

The RAD Studio IDE offers two application UI models:

- Single Document Interface (SDI)
- Multiple Document Interface (MDI)

Single Document Interface

A SDI application normally contains a single document view.

Multiple Document Interface

In an MDI application, more than one document or child window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors.

MDI applications require more planning and are more complex to design than SDI applications. MDI applications spawn child windows that reside within the client window; the main form contains child forms. For instance, you need to set the FormStyle property of the TForm object to specify whether a form is a child (fsMDIChild) or main form (fsMDIForm). It is a best practice to define a base class for your child forms and derive each child form from this class. Otherwise, you will have to reset the form properties of the child. MDI applications often include a Window pop-up on the main menu that has items such as Cascade and Tile for viewing multiple windows in various styles. When a child window is minimized, its icon is located in the MDI parent form.

Console Applications

Console applications are 32-bit programs that run in a console window without a graphical interface. These applications typically do not require much user input and perform a limited set of functions. Any application that contains `{$APPTYPE CONSOLE}` in the code opens a console window of its own.

Service Applications

Service applications take requests from client applications, process those requests, and return the information to the client applications. Service applications typically run in the background without much user input. A Web, FTP, or an email server is an example of a service application.

Creating Packages and DLLs

Dynamic link libraries (DLLs) are modules of compiled code that work in conjunction with an executable to provide functionality to an application. You can create DLLs in cross-platform programs.

Packages are special DLLs used by Delphi applications, the IDE, or both. The two types of packages are runtime and designtime. Runtime packages provide functionality to a program while that program is running. Designtime packages extend the functionality of the IDE.

For most applications, packages provide greater flexibility and are easier to create than DLLs. However,here are a few situations where DLLs would work better than packages:

- Your code module will be called from non-Delphi applications.
- You are extending the functionality of a Web server.
- You are creating a code module to be used by third-party developers.
- Your project is an OLE container.

You cannot pass Delphi runtime type information (RTTI) across DLLs or from a DLL to an executable. If you pass an object from one DLL to another DLL or to an executable, you will not be able to use the **is** or **as** operators with the passed object. This is because the **is** and **as** operators need to compare RTTI. If you need to pass objects from a library, use packages instead of DLLs, because packages can share RTTI. Similarly, you should use packages instead of DLLs in Web Services because they rely on Delphi RTTI.

See Also

Building a Windows Application (see page 89)

Building a Windows “Hello World” Console Application (see page 88)

Building a VCL Forms “Hello World” Application (see page 101)

2 Procedures

This section provides how-to information for various areas of RAD Studio development.

Topics

Name	Description
CodeGuard Procedures (see page 48)	Use these procedures to debug your C++ applications with CodeGuard.
Database Procedures (see page 50)	This topic describes how to use the database components in the **Tool Palette**, like **dbGo** components, **dbExpress** components, **BDE** components, and **DataSnap** components.
Interoperable Applications Procedures (see page 79)	Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. The simplest COM objects are servers that expose properties and methods (and possibly events) through a default interface that clients can call. The **COM Object Wizard** builds a lightweight COM object whose default interface descends from IUnknown or that implements an interface already registered on your system. This wizard provides the most flexibility in the types of COM objects you can create.
Reporting Procedures (see page 81)	This topic provides how-to information on using reporting solutions.
VCL Procedures (see page 82)	This section provides how-to information on developing VCL for Win32 applications.
WebSnap Procedures (see page 147)	This section provides how-to information on developing WebSnap applications. Please note that WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.
Web Services Procedure (see page 153)	This section provides how-to information on developing and using web services.

2.1 CodeGuard Procedures

Use these procedures to debug your C++ applications with CodeGuard.

Topics

Name	Description
Using CodeGuard (see page 48)	This procedure describes how to use CodeGuard when debugging a C++ application.

2.1.1 Using CodeGuard

This procedure describes how to use CodeGuard when debugging a C++ application.

To run a C++ application with CodeGuard reporting

1. Enable the CodeGuard reporting tool.
2. Enable CodeGuard compiler options for your project.
3. Choose **Run** ▶ **Run** to run your application.

During the execution of your application, CodeGuard runtime errors appear in the **Message** view.

CodeGuard also generates an error log named `<project_name>.cgl` that lists any errors it finds. The error log is located in the same directory as your executable.

Note: If you suspect that your program accesses a freed memory block but CodeGuard does not report an error, increase the value of Maximum memory block size

or **Delay queue length** on the **Resource Options** page of the **Configure CodeGuard** dialog box.

To enable the CodeGuard reporting tool

1. Choose **Tools** ▶ **CodeGuard Configuration** to display the **CodeGuard Configuration** dialog box.
2. Verify that CodeGuard is enabled.
3. Click **OK**.

Note: If you change any CodeGuard settings in the CodeGuard Configuration

dialog box, CodeGuard generates a `.cgi` configuration file with the same name and directory as your project file.

To enable CodeGuard compiler options for your project

1. Choose **Project** ▶ **Options** ▶ **C++ Compiler** ▶ **Debugging** to display the CodeGuard compiler options.
2. Check **Enable all CodeGuard options** to enable full CodeGuard coverage.
3. Click **OK**.
4. Rebuild your project.

Note: If you compile and link your project in separate steps, remember to include the CodeGuard library (cg32.lib

) before including other libraries.

See Also

CodeGuard Overview (see page 9)

CodeGuard Configuration dialog box

2.2 Database Procedures

This topic describes how to use the database components in the **Tool Palette**, like **dbGo** components, **dbExpress** components, **BDE** components, and **DataSnap** components.

Topics

Name	Description
Accessing Schema Information (see page 51)	The schema information or metadata includes information about what tables and stored procedures are available on the server and the information about these tables and stored procedures (like the fields of a table, the indexes that are defined, and the parameters a stored procedure uses).
Using Data Explorer to Obtain Connection Information (see page 52)	Before you have a connection, you can use **Data Explorer** to assemble connection strings.
Configuring TSQL Connection (see page 53)	The first step when working with a unidirectional dataset is to connect it to a database server. At designtime, once a dataset has an active connection to a database server, the **Object Inspector** can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the **Object Inspector** can list what stored procedures are available on the server. The connection to a database server is represented by a separate **TSQLConnection** component. You work with **TSQLConnection** like any other database connection component.
Connecting to the Application Server using DataSnap Components (see page 55)	A client application uses one or more connection components in the **DataSnap** category of the **Tool Palette** to establish and maintain a connection to an application server.
Debugging dbExpress Applications using TSQLMonitor (see page 55)	While you are debugging your database application, you can monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the dbExpress driver).
Executing the Commands using TSQLDataSet (see page 56)	You can use a unidirectional dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements. The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it, but pass the command to the server for execution.
Fetching the Data using TSQLDataSet (see page 57)	
Specifying the Data to Display using TSQLDataSet (see page 58)	
Specifying the Provider using TLocalConnection or TConnectionBroker (see page 59)	Client datasets are specialized datasets that hold all the data in memory. They use a provider to supply them with data and apply updates when they cache updates from a database server or another dataset, represent the data in an XML document, and store the data in the client portion of a multi-tiered application.
Using BDE (see page 60)	
Using DataSnap (see page 60)	A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits, such as centralized business logic and thin client applications. Multi-tiered applications use the components on the **DataSnap** category in the **Tool Palette**. DataSnap provides multi-tier database capability to Delphi applications by allowing client applications to connect to providers in an application server.
Using TBatchMove (BDE) (see page 61)	**TBatchMove** copies a table structure or its data. It can be used to move entire tables from one database format to another.

Connecting to Databases with TDatabase (see page 62)	**TDatabase** sets up a persistent connection to a database, especially a remote database requiring a user login and password. **TDatabase** is especially important because it permits control over database transaction processing with the BDE when connected to a remote SQL database server. Use **TDatabase** when a BDE-based database application requires: • Persistent database connections • Customized database server logins • Transaction control • Application-specific BDE aliases
Using TQuery (Procedure) (see page 63)	**TQuery** is a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records.
Using TSQLQuery (see page 65)	**TSQLQuery** represents a query that is executed using dbExpress. **TSQLQuery** can represent the results of a SELECT statement or perform actions on the database server using statements such as INSERT, DELETE, UPDATE, ALTER TABLE, and so on. You can add a **TSQLQuery** component to a form at design time, or create one dynamically at runtime.
Using TSQLStoredProc (Procedure) (see page 66)	**TSQLStoredProc** represents a stored procedure that is executed using dbExpress. **TSQLStoredProc** can represent the result set if the stored procedure returns a cursor. You can add a **TSQLStoredProc** component to a form at design time, or create one dynamically at runtime.
Using TSQLTable (see page 66)	**TSQLTable** represents a database table that is accessed using dbExpress. **TSQLTable** generates a query to fetch all of the rows and columns in a table you specify. You can add a **TSQLTable** component to a form at designtime, or create one dynamically at runtime.
Managing Database Sessions Using TSession (see page 67)	A session provides global connection over a group of database components. A default **TSession** component is automatically created for each database application. You must use **TSession** component only if you are creating a multithreaded database application. Each database thread requires its own session components.
Using TSimpleDataSet (see page 68)	**TSimpleDataSet** is a special type of client dataset designed for simple two-tiered applications. Like a unidirectional dataset, it can use an SQL connection component to connect to a database server and specify an SQL statement to execute on that server. Like other client datasets, it buffers data in memory to allow full navigation and editing support.
Using TSimpleObjectBroker (see page 69)	If you have multiple COM-based servers that your client application can choose from, you can use an Object Broker to locate an available server system.
Using TStoredProc (see page 69)	TStoredProc is a stored procedure-type dataset that executes a stored procedure that is defined on a database server.
Using TTable (Procedure) (see page 70)	TTable is a table-type dataset that represents all of the rows and columns of a single database table.
Using TUpdateSQL to Update a Dataset (see page 72)	When the BDE-enabled dataset represents a stored procedure or a query that is not “live”, it is not possible to apply updates directly from the dataset. Such datasets may also cause a problem when you use a client dataset to cache updates.
Using dbExpress (see page 72)	
Adding a New Connection to the Data Explorer (see page 73)	You can add new connections to the **Data Explorer**, which persist as long as the connection object exists.
Browsing a Database in the Data Explorer (see page 74)	Once you have a live connection, you can use the **Data Explorer** to browse database objects.
Executing SQL in the Data Explorer (see page 74)	You can write, edit, and execute SQL in an **SQL Window**, which is available from within the **Data Explorer**.
Modifying Connections in the Data Explorer (see page 75)	You can modify connections in a variety of ways from the **Data Explorer**.
Connecting to a Database using the dbExpress Driver Framework (see page 76)	This procedure tells how to use the dbExpress driver framework to connect to a database and read its records. In the sample code, the dbExpress ini files contain all the information about the particular database connection, such as driver, user name, password, and so on.

2.2.1 Accessing Schema Information

The schema information or metadata includes information about what tables and stored procedures are available on the server

and the information about these tables and stored procedures (like the fields of a table, the indexes that are defined, and the parameters a stored procedure uses).

To access schema information

1. To populate a unidirectional dataset with metadata from the database server, call SetSchemaInfo method to indicate what data you want to see.
2. Set the type of schema information parameter of SetSchemaInfo method.
3. Set the name of table or stored procedure parameter of SetSchemaInfo method.
4. To fetch data after using the dataset for metadata, do one of the following:

- Set the CommandText property to specify the query, table, or stored procedure from which you want to fetch data.
- Set the type of schema information to stNoSchema and call SetSchemaInfo method.

  Note: If you choose the second option, the dataset fetches the data specified by the CommandText property.

See Also

dbExpress Components (see page 16)

Configuring TSQLConnection (see page 53)

Specifying Display Data (see page 58)

Fetching the Data (see page 57)

Executing the Commands (see page 56)

Debugging dbExpress Applications (see page 55)

Using TSQLTable (see page 66)

Using TSQLStoredProc (see page 66)

Using TSQLQuery (see page 65)

Using TSimpleDataSet (see page 68)

Using Unidirectional Datasets (see page 1823)

2.2.2 Using Data Explorer to Obtain Connection Information

Before you have a connection, you can use **Data Explorer** to assemble connection strings.

To browse database providers

1. Choose **View** ▶ **Data Explorer** or click the **Data Explorer** tab in the **Project Manager**.
2. Expand a provider node under **dbExpress** to expose the list of available database providers.
3. Expand the second–level provider node to view the list of potential connections.

To get connection information

1. In the **Data Explorer** list of providers, right-click a second–level provider.
2. In the context menu, click Modify Connection. The **Modify Connection** dialog is displayed.
3. In the **Modify Connection** dialog, click **Advanced**. The **Advanced Properties** dialog is displayed, containing a scrollable list

of connection properties. The left column contains connection string keys; the right column contains their values.

To format a connection string

1. In the **Advanced Properties** dialog, enter values in the right-hand column for the connection properties named in the left column.
2. The text field at the bottom of the **Advanced Properties** dialog contains the resulting connection string text. Drag the cursor over the connection string text to select the connection string text. Copy the text. You can then paste this connection string wherever you want.

See Also

Browsing a Database in the Data Explorer (see page 74)

Adding a New Connection (see page 73)

Modifying Connections (see page 75)

2.2.3 Configuring TSQL Connection

The first step when working with a unidirectional dataset is to connect it to a database server. At designtime, once a dataset has an active connection to a database server, the **Object Inspector** can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the **Object Inspector** can list what stored procedures are available on the server. The connection to a database server is represented by a separate **TSQLConnection** component. You work with **TSQLConnection** like any other database connection component.

Metaprocedure: Configuring a TSQL Connection

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLConnection** component to the form.
4. Identify the driver (see steps below).
5. Specify connection parameters (see steps below).
6. Identify a database connection (see steps below).
7. Display and use the **dbExpress Connection Editor** (see steps below).

To identify the driver

1. Select the **TSQLConnection** component.
2. In the **Object Inspector**, set the DriverName property, to an installed dbExpress driver.
3. Identify the files associated with the driver name. Select any of the following:

- The dbExpress driver
- The dynamic link library

Note: The relationship between the dbExpress driver or dynamic link library and the database name is stored in a file called dbxdrivers.ini

, which is updated when you install a dbExpress driver. The SQL connection component looks the dbExpress driver and the dynamic-link library up in `dbxdrivers.ini` when given the value of DriverName. When you set the DriverName property, **TSQLConnection** automatically sets the LibraryName and VendorLib properties to the names of the associated dlls. Once LibraryName and VendorLib have been set, your application does not need to rely on `dbxdrivers.ini`.

To specify a connection parameter

1. Double-click on the Params property in the **Object Inspector** to edit the parameters using **Value List Editor** at designtime.
2. Use the Params.Values property to assign values to individual parameters at run time.

To identify a database connection

1. Set the ConnectionName property to a valid connection name. This automatically sets the DriverName and Params properties.
2. Edit the Params property to change the saved set of parameter values.
3. Set the LoadParamsOnConnect property to **True** to develop your application using one database and deploy it using another. This causes **TSQLConnection** to automatically set DriverName and Params to the values associated with ConnectionName in `dbxconnections.ini` when the connection is opened.
4. Call the LoadParamsFromIniFile method. This method sets DriverName and Params to the values associated with ConnectionName in `dbxconnections.ini` (or in another file that you specify). You might choose to use this method if you want to then override certain parameter values before opening the connection.

To display the Connection Editor

1. Double-click the **TSQLConnection** component. The **dbExpress Connection Editor** appears, with a drop-down drivers list, a list of connection names for the currently selected driver, and a connection parameters table for the currently selected connection name.
2. From the **Driver Name** drop-down list, select a **driver** to indicate the connection to use.
3. From the **Connection Name** list, select a **connection name**.
4. Choose the configuration that you want.
5. Click the **Test Connection** button to check for a valid configuration.

To define and modify connections using the Connection Editor

1. To edit the currently selected named connections in `dbxconnections.ini`, edit the parameter values in the parameter table.
2. Click **OK**. The new parameter values are saved to `dbxconnections.ini`.
3. Click the **Add Connection** button to define a new connection. The **New Connection** dialog appears.
4. In the **New Connection** dialog box, set the **Driver Name** and the **Connection Name**.
5. Click **OK**.
6. Click the **Delete Connection** button to delete the currently selected named connection from `dbxconnections.ini`.
7. Click the **Rename Connection** button to change the name of the currently selected named connection.

See Also

dbExpress Components (see page 16)

Specifying Display Data (see page 58)

Fetching the Data (see page 57)

Executing the Commands (see page 56)

Accessing Schema Information (see page 51)

Debugging dbExpress Applications (see page 55)

Using TSQLTable (see page 66)

Using TSQLStoredProc (see page 66)

Using TSQLQuery (see page 65)

Using TSimpleDataSet (see page 68)

Using Unidirectional Datasets (see page 1823)

2.2.4 Connecting to the Application Server using DataSnap Components

A client application uses one or more connection components in the **DataSnap** category of the **Tool Palette** to establish and maintain a connection to an application server.

To connect to the application server using DataSnap components

1. Identify the protocol for communicating with the application server.
2. Locate the server machine.
3. Identify the application server on the server machine.
4. If you are not using SOAP, identify the server using the ServerName or ServerGUID property.
5. Manage server connections.

See Also

Connecting to the Application Server (see page 1517)

Using TLocalConnection or TConnectionBroker (see page 59)

Using TSimpleObjectBroker (see page 69)

2.2.5 Debugging dbExpress Applications using TSQLMonitor

While you are debugging your database application, you can monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the dbExpress driver).

To debug dbExpress applications

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. To monitor SQL commands, from the **dbExpress** category of the **Tool Palette**, drag a **TSQLMonitor** component to the form.
4. Set the SQLConnection property of the **TSQLMonitor** to the **TSQLConnection** component.
5. Set the Active property of the **TSQLMonitor** to **True**.

To use a callback to monitor SQL commands

1. Use the SetTraceEvent method of the **TSQLConnection** component.
2. Set the TDBXTraceEvent event parameter.

The dbExpress driver triggers the event every time the SQL connection component passes a command to the server or the

server returns an error message.

Warning: Do not call SetTraceEvent if the TSQLConnection

object has an associated **TSQLMonitor** component. **TSQLMonitor** uses the callback mechanism to work, and **TSQLConnection** can only support one callback at a time.

See Also

dbExpress Components (see page 16)

Configuring TSQLConnection (see page 53)

Specifying Display Data (see page 58)

Fetching the Data (see page 57)

Executing the Commands (see page 56)

Accessing Schema Information (see page 51)

Using Unidirectional Datasets (see page 1823)

2.2.6 Executing the Commands using TSQLDataSet

You can use a unidirectional dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements. The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it, but pass the command to the server for execution.

To execute commands

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLDataSet** component to the form.
4. Specify the command to execute.
5. Execute the command.
6. Create and modify server metadata.

To specify the command to execute

1. Set the CommandType and CommandText properties in the **Object Inspector** to specify the command for a **TSQLDataSet**.
2. Set the SQL property in the **Object Inspector** to specify the SQL statement to pass to the server for a **TSQLQuery** .
3. Set the StoredProcName property in the **Object Inspector** to specify the name of the stored procedure to execute for a **TSQLStoredProc** .

To execute the command

1. If the dataset is an instance of a **TSQLDataSet** or a **TSQLQuery**, call the ExecSQL method.
2. If the dataset is an instance of a **TSQLStoredProc**, call the ExecProc method.

   Tip: If you are executing the query or stored procedure multiple times, it is a good idea to set the Prepared property to **True**.

To create and modify server metadata

1. To create tables in a database, use the CREATE TABLE statement.
2. To create new indexes for those tables, use the CREATE INDEX statement.
3. To add various metadata objects, use CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, and CREATE PROCEDURE statements.
4. To delete any of the above metadata objects, use DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, and DROP PROCEDURE.
5. To change the structure of a table, use the ALTER TABLE statement.

See Also

dbExpress Components (see page 16)

Configuring TSQLConnection (see page 53)

Specifying Display Data (see page 58)

Fetching the Data (see page 57)

Accessing Schema Information (see page 51)

Debugging dbExpress Applications (see page 55)

Using TSQLTable (see page 66)

Using TSQLStoredProc (see page 66)

Using TSQLQuery (see page 65)

Using Unidirectional Datasets (see page 1823)

2.2.7 Fetching the Data using TSQLDataSet

To fetch the data

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLDataSet** component to the form.
4. To fetch the data for a unidirectional dataset, do one of the following:

- In the **Object Inspector**, set the Active property to **True**.
- Call the Open method at runtime.

   Tip: Use GetMetadata property to selectively fetch metadata on a database object. Set GetMetadata to False if you are fetching a dataset for read-only purposes.

5. Set its Prepared property to **True** to prepare the dataset explicitly.
6. Call the NextRecordSet method to fetch multiple sets of records.

   Note: NextRecordSet returns a newly created TCustomSQLDataSet component that provides access to the next set of records. That is, the first time you call NextRecordSet, it returns a dataset for the second set of records. Calling NextRecordSet returns a third dataset, and so on, until there are no more sets of records. When there are no additional datasets, NextRecordSet does not return anything.

See Also

dbExpress Components (see page 16)

Specifying Display Data (see page 58)

Fetching the Data

Executing the Commands (see page 56)

Accessing Schema Information (see page 51)

Debugging dbExpress Applications (see page 55)

Using TSQLTable (see page 66)

Using TSQLStoredProc (see page 66)

Using TSQLQuery (see page 65)

Using Unidirectional Datasets (see page 1823)

2.2.8 Specifying the Data to Display using TSQLDataSet

To specify the data to display

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLDataSet** component to the form.
4. For **TSQLDataSet**, specify the type of unidirectional dataset by CommandType property in the **Object Inspector**.
5. Specify whether information comes from results of query, a database table, or a stored procedure.

To display results from a query

1. Set the CommandType property to ctQuery for a **TSQLDataSet**.
2. For **TSQLQuery**, drag a **TSQLQuery** component from the **Tool Palette** to the form.
3. Set the SQL property to the query you want to assign.
4. Select **TSQLDataSet**.
5. Click the CommandText property. The **CommandText Editor** opens.
6. In the **CommandText Editor**, set the SQL property to the text of the query statement.

   Note: When you specify the query, it can include parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values that appear in the SQL statement. SQL defines queries such as UPDATE queries that perform actions on the server but do not return records.

To display records in a table

1. In the **Object Inspector**, set the CommandType property to ctTable. **TSQLDataSet** generates a query based on the values of two properties: CommandText that specifies the name of the database table that the **TSQLDataSet** object should represent and SortFieldNames that lists the names of any fields to use to sort the data, in the order of significance
2. Drag a **TSQLTable** component to the form.
3. In the **Object Inspector** , set the TableName property to the table you want.
4. Set the IndexName property to the name of an index defined on the server or set the IndexFieldNames property to a

semicolon-delimited list of field names to specify the order of fields in the dataset.

To display the results of a stored procedure

1. In the **Object Inspector**, set the CommandType property to ctStoredProc.
2. Specify the name of the stored procedure as the value of the CommandText property.
3. Set the StoredProcName property to the name of the stored procedure for **TSQLStoredProc**.

   Note: After you have identified a stored procedure, your application may need to enter values for any input parameters of the stored procedure or retrieve the values of output parameters after you execute the stored procedure.

See Also

dbExpress Components (see page 16)

Configuring TSQLConnection (see page 53)

Fetching the Data (see page 57)

Executing the Commands (see page 56)

Accessing Schema Information (see page 51)

Debugging dbExpress Applications (see page 55)

Using TSQLTable (see page 66)

Using TSQLStoredProc (see page 66)

Using TSQLQuery (see page 65)

Using TSimpleDataSet (see page 68)

Using Unidirectional Datasets (see page 1823)

2.2.9 Specifying the Provider using TLocalConnection or TConnectionBroker

Client datasets are specialized datasets that hold all the data in memory. They use a provider to supply them with data and apply updates when they cache updates from a database server or another dataset, represent the data in an XML document, and store the data in the client portion of a multi-tiered application.

To specify the provider

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. From the **DataSnap** category of the **Tool Palette**, drag a **TConnectionBroker** component to the form if the provider is on a remote application server.
4. In the **Object Inspector**, set the ConnectionBroker property of your client dataset to the **TConnectionBroker** component to the form.
5. From the **DataSnap** category of the **Tool Palette**, drag a **TLocalConnection** component to the form if the provider is in the same application as the client dataset.
6. Set the RemoteServer property of your client dataset to the **TLocalConnection** component to the form.

See Also

Using DataSnap (see page 60)

Connecting To Application Server (see page 55)

Using TSimpleObjectBroker (see page 69)

2.2.10 Using BDE

To use BDE

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog box opens.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. From the **BDE** category of the **Tool Palette**, drag a **TTable** component to the form. This will encapsulate the full structure of data in an underlying database table.
4. From the **BDE** category of the **Tool Palette**, drag a **TQuery** component to the form. This will encapsulate an SQL statement and enables applications to access the resulting records.
5. From the **BDE** category of the **Tool Palette**, drag a **TStoredProc** component to the form. This will execute a stored procedure that is defined on a database server.
6. From the **BDE** category of the **Tool Palette**, drag a **TBatchMove** component to the form. This will copy a table structure or its data.
7. From the **BDE** category of the **Tool Palette**, drag a **TUpdateSQL** component to the form. This will provide a way to update the underlying datasets.

See Also

BDE Overview (see page 15)

Using TDatabase (see page 62)

Using TSession (see page 67)

Using TTable (see page 70)

Using TQuery (see page 63)

Using TStoredProc (see page 69)

Using TBatchMove (see page 61)

Using TUpdateSQL (see page 72)

2.2.11 Using DataSnap

A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits, such as centralized business logic and thin client applications.

Multi-tiered applications use the components on the **DataSnap** category in the **Tool Palette**. DataSnap provides multi-tier database capability to Delphi applications by allowing client applications to connect to providers in an application server.

To build multi-tiered database applications using DataSnap

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. From the **DataSnap** category of the **Tool Palette**, drag a **TDCOMConnection** component to the form. This will establish a DCOM connection to a remote server in a multi-tiered database application.
4. From the **DataSnap** category of the **Tool Palette**, drag a **TSocketConnection** component to the form. This will establish a TCP/IP connection to a remote server in a multi-tiered database application.
5. From the **DataSnap** category of the **Tool Palette**, drag a **TSimpleObjectBroker** component to the form. This will locate a server for a connection component from a list of available application servers.
6. From the **DataSnap** category of the **Tool Palette**, drag a **TWebConnection** component to the form. This will establish an HTTP connection to a remote server in a multi-tiered database application.
7. From the **DataSnap** category of the **Tool Palette**, drag a **TConnectionBroker** component to the form. This will centralize all connections to the application server so that applications do not need major rewriting when changing the connection protocol.
8. From the **DataSnap** category of the **Tool Palette**, drag a **TSharedConnection** component to the form. This will connect to a child remote data module when the application server is built using multiple remote data modules.
9. From the **DataSnap** category of the **Tool Palette**, drag a **TLocalConnection** component to the form. This will provide access to IAppServer methods that would otherwise be unavailable, and make it easier to scale up to a multi-tiered application at a later time. It acts like a connection component for providers that reside in the same application.

See Also

Connecting To Application Server (see page 55)

Deploying Multi-tiered Database Applications (DataSnap) (see page 1952)

Using TLocalConnection or TConnectionBroker (see page 59)

Using TSimpleObjectBroker (see page 69)

2.2.12 Using TBatchMove (BDE)

TBatchMove copies a table structure or its data. It can be used to move entire tables from one database format to another.

To use TBatchMove

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. Create a batch move component.
4. Specify a batch move mode.
5. Map data types.
6. Execute a batch move.
7. Handle batch move errors.

See Also

Using TBatchMove (see page 1683)

Using TTable (see page 70)

Using TQuery (see page 63)

Using TStoredProc (see page 69)

Using Update Objects to Update a Dataset (see page 1692)

Using TSession (see page 67)

2.2.13 Connecting to Databases with TDatabase

TDatabase sets up a persistent connection to a database, especially a remote database requiring a user login and password. **TDatabase** is especially important because it permits control over database transaction processing with the BDE when connected to a remote SQL database server. Use **TDatabase** when a BDE-based database application requires:

- Persistent database connections
- Customized database server logins
- Transaction control
- Application-specific BDE aliases

To connect to databases with TDatabase

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. Associate a database component with a session.
4. Identify the database.
5. Open a connection using TDatabase.

To associate a database component with a session

1. From the **BDE** category of the **Tool Palette**, drag a **TDatabase** component to the form.
2. Drag a **TSession** component to the form.
3. In the **Object Inspector**, set the SessionName property of the **TSession** component. SessionName is set to “Default," which means it is associated with the default session component that is referenced by the global Session variable.
4. Add a **TSession** component for each session if you use multiple sessions.
5. Set the SessionName property of the **TDatabase** component to the SessionName property of the **TSession** component to associate your dataset with a session component.
6. Read the Session property to access the session component with which the database is associated at runtime. If SessionName is blank or “Default," the Session property references the same **TSession** instance referenced by the global Session variable.

Session enables applications to access the properties, methods, and events of a database component’s parent session component without knowing the session’s actual name. If you are using an implicit database component, the session for that database component is the one specified by the dataset’s SessionName property.

To identify the database

1. In the drop-down lists for dataset components, specify the alias name or the name of an existing BDE alias for a database component.

   Note: This clears any value already assigned to DriverName. Alternatively, you can specify a driver name instead of an alias when you create a local BDE alias for a database component using the DatabaseName property. Specifying the driver name

clears any value already assigned to AliasName. To provide your own name for a database connection, set the DatabaseName. To specify a BDE alias at designtime, assign a BDE driver.

2. Create a local BDE alias.
3. Double-click a database component. The **Database** editor opens.
4. In the **Name** edit box in the properties editor, enter the same name as specified by the DatabaseName property.
5. In the **Alias name** combo box, enter an existing BDE alias name or choose from existing aliases in the drop-down list.
6. To create or edit connection parameters at designtime, do one of the following:

- Use the Database Explorer or BDE Administration utility.
- Double-click the Params property in the **Object Inspector** to invoke the **Value List** editor.
- Double-click a database component in a data module or form to invoke the **Database** editor.

Note: All of these methods edit the Params property for the database component. When you first invoke the Database Properties

editor, the parameters for the BDE alias are not visible. To see the current settings, click **Defaults**. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click **Clear**. Changes you make take effect only when you click **OK**.

To open a connection using TDatabase

1. In the Params property of a **TDatabase** component, configure the ODBC driver for your application.
2. To connect to a database using **TDatabase**, set the Connected property to **True** or call the Open method.

Note: Calling TDatabase. Rollback does not call TDataSet. Cancel for any data sets associated with the database.

See Also

BDE Overview (see page 15)

Using TSession (see page 67)

Using TTable (see page 70)

Using TQuery (see page 63)

Using TStoredProc (see page 69)

Using TBatchMove (see page 61)

Using TUpdateSQL (see page 72)

2.2.14 Using TQuery (Procedure)

TQuery is a query-type dataset that encapsulates an SQL statement and enables applications to access the resulting records.

To use TQuery

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. Associate the dataset with database and session connections.
4. Create heterogeneous queries.

5. Obtain an editable result set.
6. Update read-only result sets.

To associate a dataset with database and session connections

1. From the **BDE** category of the **Tool Palette**, drag a **TDatabase** component to the form.
2. Drag a **TSession** component to the form.
3. Set the DatabaseName property of the **TDatabase** component to associate a BDE-enabled dataset with a database. For the **TDatabase** component, database name is the value of the DatabaseName property of the database component.
4. Specify a BDE alias as the value of DatabaseName if you want to use an implicit database component and the database has a BDE alias.

   Note: A BDE alias represents a database plus configuration information for that database. The configuration information associated with an alias differs by database type (Oracle, Sybase, InterBase, Paradox, dBASE, and so on).

5. In the **Object Inspector**, set the DatabaseName to specify the directory where the database tables are located if you want to use an implicit database component for a Paradox or dBASE database.
6. Use the default session to control all database connections in your application.
7. Set the SessionName property of the **TSession** component to associate your dataset with an explicitly created session component .

   Note: Whether you use the default session or explicitly specify a session using the SessionName property, you can access the session associated with a dataset by reading the DBSession property. If you use a session component, the SessionName property of a dataset must match the SessionName property for the database component with which the dataset is associated.

To create mixed queries

1. Define separate BDE aliases for each database accessed in the query using the BDE Administration tool or the SQL explorer.
2. Leave the DatabaseName property of the **TQuery** component blank. The names of the databases used will be specified in the SQL statement.
3. Set the SQL property to the SQL statement you want to execute.
4. Precede each table name in the statement with the BDE alias for the database of the table, enclosed in colons. This whole reference is then enclosed in quotation marks.
5. Set the Params property to any parameters for the query.
6. Write a Prepare method to prepare the query for execution prior to executing it for the first time.
7. Write an Open or ExecSQL method depending on the type of query you are executing.
8. Use a **TDatabase** component as an alternative to using a BDE alias to specify the database in a mixed query.
9. Configure the **TDatabase** to the database, set the TDatabase. DatabaseName to an unique value, and use that value in the SQL statement instead of a BDE alias name.

To obtain an editable result set

1. Set RequestLive property of the **TQuery** component to **True**.
2. If the query contains linked fields, treat the result set as a read-only result set, and update it.

If an application requests a live result set, but the SELECT statement syntax does not allow it, the BDE returns either a read-only result set for queries made against Paradox or dBASE, or an error code for SQL queries made against a remote server.

To update read-only result sets

1. If all updates are applied to a single database table, indicate the underlying table to update in an OnGetTableName event handler.

2. Set the query's UpdateObject property to the **TUpdateSQL** object you are using to have more control over applying updates.
3. Set the DeleteSQL, InsertSQL, and ModifySQL properties of the update object to the SQL statements that perform the appropriate updates for your query's data.

If you are using the BDE to cache updates, you must use an update object.

See Also

BDE Overview (see page 15)

Using TDatabase (see page 62)

Using TSession (see page 67)

Using TTable (see page 70)

Using TStoredProc (see page 69)

Using TBatchMove (see page 61)

Using TUpdateSQL (see page 72)

2.2.15 Using TSQLQuery

TSQLQuery represents a query that is executed using dbExpress. **TSQLQuery** can represent the results of a SELECT statement or perform actions on the database server using statements such as INSERT, DELETE, UPDATE, ALTER TABLE, and so on. You can add a **TSQLQuery** component to a form at design time, or create one dynamically at runtime.

To use TSQLQuery

1. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLQuery** component to the form.
2. In the **Object Inspector**, set its Name property to a unique value appropriate to your application.
3. Set the SQLConnection property.
4. Click the ellipsis button next to the SQL property of the **TSQLQuery** component. The **String List** editor opens.
5. In the **String List** editor, type the query statement you want to execute.
6. If the query data is to be used with visual data controls, add a data source component to the form.
7. Set the DataSet property of the data source component to the query-type dataset.
8. To activate the query component, set the Active property to **True** or call the Open method at runtime.

See Also

dbExpress Components (see page 16)

Configuring TSQLConnection (see page 53)

Specifying Display Data (see page 58)

Fetching the Data (see page 57)

Executing the Commands (see page 56)

Accessing Schema Information (see page 51)

Using TSQLTable (see page 66)

Using TSQLStoredProc (see page 66)

Using TSimpleDataSet (see page 68)

Using Unidirectional Datasets (see page 1823)

2.2.16 Using TSQLStoredProc (Procedure)

TSQLStoredProc represents a stored procedure that is executed using dbExpress. **TSQLStoredProc** can represent the result set if the stored procedure returns a cursor. You can add a **TSQLStoredProc** component to a form at design time, or create one dynamically at runtime.

To use TSQLStoredProc

1. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLStoredProc** component to the form.
2. In the **Object Inspector**, set its Name property to a unique value appropriate to your application.
3. Set the SQLConnection property.
4. Set the StoredProcName property to specify the stored procedure to execute.
5. If the stored procedure returns a cursor to be used with visual data controls, add a data source component to the form.
6. Set the DataSet property of the data source component to the stored procedure-type dataset.
7. Provide input parameter values for the stored procedure, if necessary.
8. To execute the stored procedure that returns a cursor, use the Active property or call the Open method.
9. Process any results.

See Also

dbExpress Components (see page 16)

Configuring TSQLConnection (see page 53)

Specifying Display Data (see page 58)

Fetching the Data (see page 57)

Executing the Commands (see page 56)

Accessing Schema Information (see page 51)

Using TSQLTable (see page 66)

Using TSQLQuery (see page 65)

Using TSimpleDataSet (see page 68)

Using Unidirectional Datasets (see page 1823)

2.2.17 Using TSQLTable

TSQLTable represents a database table that is accessed using dbExpress. **TSQLTable** generates a query to fetch all of the rows and columns in a table you specify. You can add a **TSQLTable** component to a form at designtime, or create one dynamically at runtime.

To use TSQLTable

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog displays.

2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. From the **dbExpress** category of the **Tool Palette**, drag a **TSQLTable** component to the form.
4. In the **Object Inspector**, set its Name property to a unique value appropriate to your application.
5. Set the SQLConnection property
6. Set the TableName property to the name of the table in the database.
7. Add a data source component to the form.
8. Set the DataSet property of the data source component to the the name of the dataset.

See Also

dbExpress Components (see page 16)

Configuring TSQLConnection (see page 53)

Specifying Display Data (see page 58)

Fetching the Data (see page 57)

Executing the Commands (see page 56)

Accessing Schema Information (see page 51)

Using TSQLStoredProc (see page 66)

Using TSQLQuery (see page 65)

Using TSimpleDataSet (see page 68)

Using Unidirectional Datasets (see page 1823)

2.2.18 Managing Database Sessions Using TSession

A session provides global connection over a group of database components. A default **TSession** component is automatically created for each database application. You must use **TSession** component only if you are creating a multithreaded database application. Each database thread requires its own session components.

To manage database sessions

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. Activate a session.
4. Specify default database connection behavior.
5. Manage database connections.
6. Work with password-protected Paradox and dBASE tables.
7. Work with BDE aliases.
8. Retrieve information about a session.
9. Create, Name, and Manage additional sessions.

See Also

Managing Database Sessions (see page 1664)

Using TTable (see page 70)

Using TQuery (see page 63)

Using TStoredProc (see page 69)

Using TDatabase (see page 62)

Using TBatchMove (see page 61)

Using TUpdateSQL (see page 72)

2.2.19 Using TSimpleDataSet

TSimpleDataSet is a special type of client dataset designed for simple two-tiered applications. Like a unidirectional dataset, it can use an SQL connection component to connect to a database server and specify an SQL statement to execute on that server. Like other client datasets, it buffers data in memory to allow full navigation and editing support.

To use TSQLStoredProc

1. From the **dbExpress** category of the **Tool Palette**, drag a **TSimpleDataSet** component to the form.
2. Set its Name property to a unique value appropriate to your application.
3. From the **dbExpress** section of the **Tool Palette**, drag a **TSQLConnection** component on the form.
4. Select **TSimpleDataSet** component. Set the Connection property to **TSQLConnection** component.
5. To fetch data from the server, do any of the following:

- Set CommandType to ctQuery and set CommandText to an SQL statement you want to execute on the server.
- Set CommandType to ctStoredProc and set CommandText to the name of the stored procedure you want to execute.
- Set CommandType to ctTable and set CommandText to the name of the database tables whose records you want to use.

6. If the stored procedure returns a cursor to be used with visual data controls, add a data source component to the form.
7. Set the DataSet property of the data source component to the **TSimpleDataSet** object.
8. To activate the dataset, use the Active property or call the Open method.
9. If you executed a stored procedure, use the Params property to retrieve any output parameters.

See Also

dbExpress Components (see page 16)

Configuring TSQLConnection (see page 53)

Specifying Display Data (see page 58)

Fetching the Data (see page 57)

Executing the Commands (see page 56)

Accessing Schema Information (see page 51)

Using TSQLTable (see page 66)

Using TSQLStoredProc (see page 66)

Using TSQLQuery (see page 65)

Using Unidirectional Datasets (see page 1823)

2.2.20 Using TSimpleObjectBroker

If you have multiple COM-based servers that your client application can choose from, you can use an Object Broker to locate an available server system.

To use TSimpleObjectBroker

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. From the **DataSnap** category of the **Tool Palette**, choose the connection component depending on the kind of connection you want.
4. From the **Tool Palette**, drag a **TSimpleObjectBroker** to the form.
5. In the **Object Inspector**, set the ObjectBroker property of the connection component that you chose in Step 3 to use this broker.

   Warning: Do not use the ObjectBroker property with SOAP connections.

See Also

Using DataSnap (see page 60)

Connecting To Application Server (see page 55)

Using TLocalConnection or TConnectionBroker (see page 59)

2.2.21 Using TStoredProc

TStoredProc is a stored procedure-type dataset that executes a stored procedure that is defined on a database server.

To use TStoredProc

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. Associate a dataset with database and session connections.
4. Bind the parameters.

To associate a dataset with database and session connections

1. From the **BDE** category of the **Tool Palette**, drag a **TDatabase** component to the form.
2. To associate a BDE-enabled dataset with a database, set the DatabaseName property. For **TDatabase** component, database name is the value of the DatabaseName property of the database component.
3. Drag a **TSession** component to the form.
4. To control all database connections in your application, use the default session.
5. In the **Object Inspector**, set the SessionName property of the **TSession** component to associate your dataset with an explicitly created session component.

   Note: If you use a session component, the SessionName property of a dataset must match the SessionName property for the

database component with which the dataset is associated.

To bind parameters

1. From the **BDE** category of the **Tool Palette**, drag a **TStoredProc** component to the form.
2. Set the ParamBindMode property to default **pbByName** to specify how parameters should be bound to the parameters on the server.
3. View the stored procedure source code of a server in the SQL Explorer if you want to set ParamBindMode to **pbByNumber**.
4. Determine the correct order and type of parameters.
5. Specify the correct parameter types in the correct order.

   Note: Some servers also support binding parameters by ordinal value, the order in which the parameters appear in the stored procedure. In this case the order in which you specify parameters in the parameter collection editor is significant. The first parameter you specify is matched to the first input parameter on the server, the second parameter is matched to the second input parameter on the server, and so on. If your server supports parameter binding by ordinal value, you can set ParamBindMode to pbByNumber

   .

See Also

BDE Overview (see page 15)

Using TTable (see page 70)

Using TQuery (see page 63)

Using TBatchMove (see page 61)

Using TUpdateSQL (see page 72)

2.2.22 Using TTable (Procedure)

TTable is a table-type dataset that represents all of the rows and columns of a single database table.

To use TTable

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **Delphi Projects** and double-click **VCL Forms Application**. The **Windows Designer** displays.
3. Associate the dataset with the database and session connections.
4. Specify the table type for local tables and control read/write access to local tables.
5. Specify a dBASE index file.
6. Rename local tables.
7. Import data from another table.

To associate a dataset with database and session connections

1. From the **BDE** category of the **Tool Palette**, drag a **TDatabase** component to the form.
2. Drag a **TSession** component to the form.
3. To associate a BDE-enabled dataset with a database, in the **Object Inspector**, set the DatabaseName property of the **TDatabase** component . For a **TDatabase** component, the database name is the value of the DatabaseName property of the database component.

4. Use the default session to control all database connections in your application.
5. Set the SessionName property of the **TSession** component to associate your dataset with an explicitly created session component.

If you use a session component, the SessionName property of a dataset must match the SessionName property for the database component with which the dataset is associated.

To specify the TableType and control read/write access

1. From the **BDE** category of the **Tool Palette**, drag a **TTable** component to the form.
2. In the **Object Inspector**, set the TableType property if an application accesses Paradox, dBASE, FoxPro, or comma-delimited ASCII text tables. BDE uses the TableType property to determine the table's type.
3. Set TableType to ttDefault if your local Paradox, dBASE, and ASCII text tables use the file extensions like, .DB, .DBF, and .TXT.
4. For other extensions, set TableType to ttParadox for Paradox, ttDBase for dBASE, ttFoxPro for FoxPro, and ttASCII for Comma-delimited ASCII text respectively.
5. Set the table component's Exclusive property to **True** before opening the table to gain sole read/write access.

   Note: If the table is already in use when you attempt to open it, exclusive access is not granted. You can attempt to set Exclusive on SQL tables, but some servers do not support exclusive table-level locking. Others may grant an exclusive lock, but permit other applications to read data from the table.

To specify a dBASE index file

1. Set the IndexFiles property to the name of the non-production index file or list the files with a .NDX extension.
2. Specify one index in the IndexName property to have it actively sorting the dataset.
3. At designtime, click the ellipsis button in the IndexFiles property. The **Index Files** editor opens.
4. To add a non-production index file or file with .NDX extension, click the **Add** button in the **Index Files** dialog and select the file from the **Open** dialog.

   Note: For each non-production index file or .NDX file, repeat Steps 3 and 4.

5. After adding all desired indexes, click the **OK** button in the **Index Files** editor.

   Note: To do steps 3-5 at runtime, access the IndexFiles property using properties and methods of string lists.

To rename local tables

1. To rename a Paradox or dBASE table at design time, right-click the table component. A drop-down context menu opens.
2. From the context menu, select **Rename Table**.
3. To rename a Paradox or dBASE table at runtime, call the table's RenameTable method.

To import data from another table

1. Use the BatchMove method of a table component to import data, copy, update, append records from another table into this table, or delete records from a table.
2. Set the name of the table from which to import data, and a mode specification that determines which import operation to perform.

See Also

BDE Overview (see page 15)

Using TDatabase (see page 62)

Using TSession (see page 67)

Using TQuery (see page 63)

Using TStoredProc (see page 69)

Using TBatchMove (see page 61)

Using TUpdateSQL (see page 72)

2.2.23 Using TUpdateSQL to Update a Dataset

When the BDE-enabled dataset represents a stored procedure or a query that is not “live”, it is not possible to apply updates directly from the dataset. Such datasets may also cause a problem when you use a client dataset to cache updates.

To update a dataset using an update object

1. From the **Tool Palette**, add a **TUpdateSQL** component to the same form as the BDE-enabled dataset.
2. In the **Object Inspector**, set the UpdateObject property of the BDE-enabled dataset component’s to the **TUpdateSQL** component in the form.
3. Set the ModifySQL, InsertSQL, and DeleteSQL properties of the update object to specify the SQL statements needed to perform updates.
4. Close the dataset.
5. Set the dataset component’s CachedUpdates property to **True** or link the dataset to the client dataset using a dataset provider.
6. Reopen the dataset.
7. Create SQL statements for update components.
8. Use multiple update objects.
9. Execute the SQL statements.

See Also

Using Update Objects to Update a Dataset (see page 1692)

Using TTable (see page 70)

Using TQuery (see page 63)

Using TStoredProc (see page 69)

Using TBatchMove (see page 61)

Using TSession (see page 67)

2.2.24 Using dbExpress

To build a database applications using dbExpress

1. Connect to the database server and configure a TSQL connection.
2. Specify the data to display.
3. Fetch the data.
4. Execute the commands.

5. Access the schema information.
6. Debug dbExpress application using **TSQLMonitor**.
7. Use **TSQLTable** to represent a table on a database server that is accessed via **TSQLConnection**.
8. Use **TSQLQuery** to execute an SQL command on a database server that is accessed via **TSQLConnection**.
9. Use **TSQLStoredProc** to execute a stored procedure on a database server that is accessed via **TSQLConnection**.

See Also

dbExpress Components (see page 16)

Configuring TSQLConnection (see page 53)

Specifying Display Data (see page 58)

Fetching the Data (see page 57)

Executing the Commands (see page 56)

Accessing Schema Information (see page 51)

Debugging dbExpress Applications (see page 55)

Using TSQLTable (see page 66)

Using TSQLStoredProc (see page 66)

Using TSQLQuery (see page 65)

Using TSimpleDataSet (see page 68)

Using dbExpress Datasets (see page 1823)

2.2.25 Adding a New Connection to the Data Explorer

You can add new connections to the **Data Explorer**, which persist as long as the connection object exists.

To add a new connection

1. Choose **View** ▸ **Data Explorer**. This displays the **Data Explorer**.
2. Select a provider from the tree list.
3. Right-click to display a pop-up menu.
4. Choose **Add New Connection**. This displays the **Add New Connection** dialog.
5. Enter the name of the new connection.
6. Click **OK**.

To modify connection settings

1. Right-click your connection and scroll down to **modify a connection**. A **Modify Connection** dialog appears.
2. Enter your connection settings and click **OK**.

See Also

Browsing a Database (see page 74)

Executing SQL in the Data Explorer (see page 74)

Modifying Connections (see page 75)

2.2.26 Browsing a Database in the Data Explorer

Once you have a live connection, you can use the **Data Explorer** to browse database objects.

To browse database objects

1. Choose **View** ▸ **Data Explorer**.
2. Expand a provider node to expose the list of available connections.
3. Expand a connection node to view the list of database objects (tables, views, and procedures).

   Note: If you receive an error because your connection is not live, you should refresh your provider, and/or modify your connection.

To retrieve data from the database

1. Expand a connection in the **Data Explorer**.
2. Double-click a table name or view name to retrieve data. This operation returns a result set into a tabbed **Data Explorer** page in the **Code Editor**.

   Tip: You can also select a table in the Data Explorer

   and right-click to display a pop-up menu with a **Retrieve Data From Table** command.

To run a stored procedure

1. Choose **View** ▸ **Data Explorer**.
2. Expand a connection in the **Data Explorer** and locate a stored procedure.
3. Double-click the stored procedure to view its parameters. The parameters open in a separate page on the design surface.
4. Edit input parameters as necessary.
5. Click the Execute button in the top left corner of the page to execute the procedure. The result set appears in a datagrid.

   Tip: You can also select a procedure in the Data Explorer

   and right-click to display a pop-up menu with an **Execute** command.

See Also

Adding a New Connection (see page 73)

Executing SQL in the Data Explorer (see page 74)

Modifying Connections (see page 75)

2.2.27 Executing SQL in the Data Explorer

You can write, edit, and execute SQL in an **SQL Window**, which is available from within the **Data Explorer**.

To open a SQL Window

1. Choose **View** ▸ **Data Explorer**.
2. Select a connection.

3. Right-click the connection and choose **SQL Window**. This opens a tabbed **SQL Window** in the **Code Editor**.

To execute SQL

1. Enter a valid SQL statement or stored procedure name in the multi-line text box at the top of the **SQL Window**.
2. Click **Execute SQL**. If the SQL statement or stored procedure is valid, the result set appears in the bottom pane of the **SQL Window**.

   Note: The SQL statement or stored procedure must operate against the current connection and its target database. You cannot execute SQL against a database to which you are not connected.

3. Click **Clear All SQL** to clear the SQL statement or stored procedure from the multi-line text box.

See Also

Browsing a Database (see page 74)

Adding a New Connection (see page 73)

Modifying Connections (see page 75)

2.2.28 Modifying Connections in the Data Explorer

You can modify connections in a variety of ways from the **Data Explorer**.

To modify connections

1. Choose **View** ▶ **Data Explorer**.
2. Select a provider.
3. Right-click to display a pop-up menu to view your options.

To refresh a connection

1. Choose **View** ▶ **Data Explorer**.
2. Select a provider.
3. Right-click to display a pop-up menu.
4. Choose **Refresh**. This operation reinitializes all connections defined for the selected provider.

To delete a connection

1. Choose **View** ▶ **Data Explorer**.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose **Delete Connection**. This displays a confirmation message that asks if you want to delete the connection.
5. Click **OK**.

To modify a connection

1. Choose **View** ▶ **Data Explorer**.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose **Modify Connection**. This displays the **Connections Editor** dialog.

5. Make changes to the appropriate values in the editor.
6. Click **OK**.

To close a connection

1. Choose **View** ▸ **Data Explorer**.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose **Close Connection**. If the connection is open, this operation closes it.

   Note: If the Close Connection

   command is disabled in the menu, the connection is not open.

To rename a connection

1. Choose **View** ▸ **Data Explorer**.
2. Select a connection.
3. Right-click to display a pop-up menu.
4. Choose **Rename Connection**. This displays **Rename Connection** dialog.
5. Enter a new name.
6. Click **OK**. The **Data Explorer** displays the connection with its new name.

See Also

Browsing a Database (see page 74)

Executing SQL in the Data Explorer (see page 74)

Adding a New Connection (see page 73)

2.2.29 Connecting to a Database using the dbExpress Driver Framework

This procedure tells how to use the dbExpress driver framework to connect to a database and read its records. In the sample code, the dbExpress ini files contain all the information about the particular database connection, such as driver, user name, password, and so on.

To connect to a database and read its records

1. Configure the connections ini file with the information about the database you are connecting to. This includes setting the driver name, user name, password, and so on.
2. Obtain a TDBXConnectionFactory, which is returned by TDBXConnectionFactory.GetConnectionFactory.
3. Get a TDBXConnectionobject returned by TDBXConnectionFactory.GetConnection.
4. Open the database connection by calling TDBXConnection.Open on the TDBXConnection instance.
5. Get a TDBXCommandobject by calling TDBXConnection.CreateCommand on the TDBXConnection instance.
6. Set the TDBXCommand's Textproperty to the desired SQL command. Call TDBXCommand.Prepare on the TDBXCommand instance.
7. Execute the SQL query by calling TDBXCommand.ExecuteQuery, which returns a TDBXReader instance.
8. Read the first database record by calling TDBXReader.Next. Call this method to retrieve successive database records.

9. Get whatever information you want from the database. For instance, TDBXReader.GetColumnCount returns the number of database columns. The TDBXReader properties ValueTypeand Value contain the data type and value for a given column number in the current record.

```
// This sample connects to a database using the ini files.
// These files must be configured for the database.
// Once connected, the sample reads values and displays the
// ANSI values for the first 100 records in a listbox.

// Get a TDBXConnection using a TDBXConnectionFactory.
// ConnectionName = section in the connections ini file.
class function TForm1.BuildConnectionFromConnectionName(
 ConnectionName: WideString): TDBXConnection;
var
 ConnectionFactory: TDBXConnectionFactory;
 ConnectionProps: TDBXProperties;
begin
 ConnectionFactory := TDBXConnectionFactory.GetConnectionFactory;
 ConnectionProps := ConnectionFactory.GetConnectionProperties(ConnectionName);
 Result := ConnectionFactory.GetConnection(ConnectionProps,
   ConnectionProps.Values[TDBXPropertyNames.UserName],
ConnectionProps.Values[TDBXPropertyNames.Password] );
end;


procedure Connect;
var
 connection: TDBXConnection;
 command: TDBXCommand;
 reader:   TDBXReader;
 value: TDBXValue;
 valueType: TDBXValueType;
 colCountStr: string;
 i, j: Integer;
 numCols: integer;
 ListBox1: TListBox;


const
 sqlCommand = 'select * from employee';

begin
 // Open connection to DB.
 connection := BuildConnectionFromConnectionName('ConnectionName');
 connection.Open;

 // Get command
 command := connection.CreateCommand();
 command.Text := sqlCommand;

 // Execute query
 command.Prepare;
 reader := command.ExecuteQuery;

 // Get values from DB
 if reader.Next then
 begin
 numCols := reader.GetColumnCount;
   Str(numCols, colCountStr);
   ListBox1.Items.Add('Number of columns = ' + colCountStr);
   j := 1;
   repeat
     for i := 0 to reader.GetColumnCount - 1 do
     begin
       valueType := reader.ValueType[i];
       if valueType.DataType = TDBXDataTypes.AnsiStringType then
```

```
        begin
          value := reader.Value[i];
          ListBox1.Items.Add(valueType.Name + ' = ' +
            value.GetString);
        end
        else
          ListBox1.Items.Add(valueType.Name);
      end;
      Inc(j);
    until (j > 100) or not reader.Next;

    reader.Next;
  end;

  // Free resources
  command.Free;
end;
```

2.3 Interoperable Applications Procedures

Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. The simplest COM objects are servers that expose properties and methods (and possibly events) through a default interface that clients can call. The **COM Object Wizard** builds a lightweight COM object whose default interface descends from IUnknown or that implements an interface already registered on your system. This wizard provides the most flexibility in the types of COM objects you can create.

Topics

Name	Description
Using COM Wizards (see page 79)	RAD Studio provides wizards that help you create COM projects and COM objects. These wizards are available for both Delphi and C++ projects. The following COM wizards are available in the list for **File ▶ New ▶ Other ▶ <personality> ▶ ActiveX**: • Active Form • Active Server Object • ActiveX Control • ActiveX Library • Automation Object • COM Object • COM+ Event Object • COM+ Subscription Object • Property Page • Transactional Object • Type Library Two other related wizards are available from the **File ▶ New ▶ Other ▶ <personality> ▶ Multitier** list: • Remote Data Module • Transactional Data Module

2.3.1 Using COM Wizards

RAD Studio provides wizards that help you create COM projects and COM objects. These wizards are available for both Delphi and C++ projects. The following COM wizards are available in the list for **File ▶ New ▶ Other ▶ <personality> ▶ ActiveX**:

- Active Form
- Active Server Object
- ActiveX Control
- ActiveX Library
- Automation Object
- COM Object

- COM+ Event Object
- COM+ Subscription Object
- Property Page
- Transactional Object
- Type Library

Two other related wizards are available from the **File** ▸ **New** ▸ **Other** ▸ **<personality>** ▸ **Multitier** list:

- Remote Data Module
- Transactional Data Module

To use a COM wizard

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog box displays.
2. In the **Item Categories** tree, click the **ActiveX** folder beneath your chosen personality, either **C++Builder Projects** or **Delphi Projects**.
3. In the **ActiveX** folder, double-click **ActiveX Library**. This creates a Dynamic Link Library [DLL] project that you can use to host in-process ActiveX Objects.
4. Choose **File** ▸ **New** ▸ **Other** again.
5. Do either one of the following:

- To display the COM wizards, click the same **ActiveX** folder that you clicked in step 2.
- To display the multitier wizards, click the **Multitier** folder in your selected personality.

6. Double-click the wizard that you want to use.

   Note: To create a client application to interact with the server created by a COM wizard, use the Import Component wizard or the TLIBIMP utility.

   If your application implements more than one COM object, you should specify the same instancing for all of them.

See Also

Overview of COM Technologies (see page 1385)

2.4 Reporting Procedures

This topic provides how-to information on using reporting solutions.

Topics

Name	Description
Adding Rave Reports to RAD Studio (see page 81)	Rave Reports offers a powerful set of tools for building reports and including them in your applications. Rave Reports are installed in a `\RaveReports` subdirectory in your installation directory. To make the Rave Reports more easily accessible, add the command executable to your **Tools** menu.

2.4.1 Adding Rave Reports to RAD Studio

Rave Reports offers a powerful set of tools for building reports and including them in your applications. Rave Reports are installed in a `\RaveReports` subdirectory in your installation directory. To make the Rave Reports more easily accessible, add the command executable to your **Tools** menu.

To add a Rave Reports command to the Tools menu

1. Choose **Tools** ▸ **Configure Tools**. This displays the **Tool Options** dialog box.
2. Click **Add**. This displays the **Tool Properties** dialog box.
3. Type `Rave Reports` in the **Title** text box.
4. Click the **Browse** button.
5. Browse to the `\RaveReports` subdirectory in your RAD Studio installation directory.
6. Select the **Rave.exe** icon.
7. Click **OK**. This adds the path for the program and the working directory to the **Tool Properties** dialog box.
8. Click **OK**
9. Click **Close**. This adds the command to your **Tools** menu that will initiate a Rave Reports session. Refer to the Rave Reports online Help for information on how to build and integrate report objects.

See Also

Rave Reports Overview (see page 35)

2.5 VCL Procedures

This section provides how-to information on developing VCL for Win32 applications.

Topics

Name	Description
Building a Windows "Hello World" Console Application (see page 88)	This "Hello World" console application demonstrates the essential steps for creating a Windows application in Delphi or C++. The application uses Windows, a console window, an event, and will display a dialog in response to a user action.
Developing a Windows Application (see page 89)	The following procedure illustrates the essential steps for building a Windows application.
Building Application Menus (see page 90)	Menus provide an easy way for your users to execute logically grouped commands. You can add or delete menu items, or drag them to rearrange them during designtime. In addition to **TMainMenu** and **TPopupMenu** components, the **Tool Palette** also contains **TActionMainMenuBar**, **TActionManager**, and **TActionToolBar**.
Building a VCL Forms Application with Decision Support Components (see page 91)	Creating a form with tables and graphs of multidimensional data consists of the following major steps: 1. Create a VCL form. 2. Add a decision query and dataset. 3. Add a decision cube. 4. Add a decision source. 5. Optionally add a decision pivot. 6. Add one or more decision grids and graphs. 7. Set the active property of the decision query (or alternate dataset component) to True.
Building VCL Forms Applications With Graphics (see page 93)	Each of the procedures listed below builds a VCL Form application that uses graphics. Build one or more of the examples and then add other graphics features to these basic VCL Form applications. 1. Draw straight lines. 2. Draw rectangles and ellipses. 3. Draw a polygon. 4. Display a bitmap image. 5. Place a bitmap in a combo box.
Building a VCL Forms MDI Application Using a Wizard (see page 93)	The VCL Forms MDI application wizard automatically creates a project that includes the basic files for an MDI application. In addition to the Main source file, the wizard creates unit files for child and about box windows, along with the supporting forms files and resources.
Building a VCL Forms MDI Application Without Using a Wizard (see page 94)	
Building a VCL Forms SDI Application (see page 96)	
Creating a New VCL Component (see page 97)	You can use the **New VCL Component** wizard to create a new VCL component for your application. The wizard detects the personality of the product you are using and creates the appropriate type of component.

Building a VCL Forms ADO Database Application (see page 98)	The following procedure describes how to build an ADO database application. Building a VCL ADO application consists of the following major steps: 1. Set up the database connection. 2. Set up the dataset. 3. Set up the data provider, client dataset, and data source. 4. Connect a DataGrid to the connection components. 5. Run the application.
Building a VCL Forms Application (see page 99)	The following procedure illustrates the essential steps to building a VCL Forms application using RAD Studio.
Creating Actions in a VCL Forms Application (see page 100)	Using RAD Studio, the following procedure illustrates how to create actions using the ActionList tool. It sets up a simple application and describes how to create a file menu item with a file open action. Building the VCL application with ActionList actions consists of the following major steps: 1. Create a main window and add tools for creating a main menu and a File open action. 2. Add the File category to the main menu. 3. Add the File open action to the File category. 4. Build and run the application.
Building a VCL Forms "Hello World" Application (see page 101)	This procedure demonstrates how to construct a simple “Hello world” VCL Forms application using either Delphi or C++. Though simple, the Windows Forms "Hello world" application demonstrates the essential steps for creating a VCL Forms application. The application uses a VCL Form, a control, an event, and displays a dialog in response to a user action. Creating the "Hello world" application consists of the following steps: 1. Create a VCL Form with a button control. 2. Write the code to display "Hello world" when the button is clicked. 3. Run the application.
Using ActionManager to Create Actions in a VCL Forms Application (see page 102)	Using RAD Studio, the following procedure illustrates how to create actions using ActionManager. It sets up a simple user interface with a text area, as would be appropriate for a text editing application, and describes how to create a file menu item with a file open action. Building the VCL application with ActionManager actions consists of the following major steps: 1. Create a main window. 2. Add a File open action to the ActionManager. 3. Create the main menu. 4. Add the action to the menu. 5. Build and run the application.
Building a VCL Forms dbExpress Database Application (see page 103)	The following procedure describes how to build a dbExpress database application. Building a VCL Forms dbExpress application consists of the following major steps: 1. Set up the database connection. 2. Set up the unidirectional dataset. 3. Set up the data provider, client dataset, and data source. 4. Connect a DataGrid to the connection components. 5. Run the application.

Building an Application with XML Components (see page 104)	This example creates a VCL Forms application that uses an XMLDocument component to display contents in an XML file.
Copying Data From One Stream To Another (see page 106)	Creating this VCL application consists of the following steps: 1. Create a project directory containing a text file to copy. 2. Create a VCL Form with a button control. 3. Write the code to read the string and write it to a file. 4. Run the application.
Copying a Complete String List (VCL) (see page 107)	Copying a string list can have the effect of appending to or overwriting an existing string list. This VCL application appends to a string list. With a simple change, it can overwrite a string list. Creating this VCL application consists of the following steps: 1. Create a VCL Form with TButtons, TComboBox, and TMemo controls. 2. Write the code to create a string list to the Button1 OnClick handler. 3. Write the code to copy the string list to the Button2 OnClick handler. 4. Run the application.
Creating Strings (see page 109)	Creating this VCL application consists of the following steps: 1. Create a VCL Form with TButton and TComboBox controls. 2. Write the code to create strings to the TButton OnClick handler. 3. Run the application.
Creating a VCL Form Instance Using a Local Variable (see page 110)	A safe way to create a unique instance of a modal VCL form is to use a local variable in an event handler as a reference to a new instance. If you use a local variable, it doesn't matter whether the form is auto-created or not. The code in the event handler makes no reference to the global form variable. Using RAD Studio, the following procedure creates a modal form instance dynamically. It (optionally) removes the second form's invocation at startup. Building this VCL application consists of the following steps: 1. Create the project directory. 2. Create two forms for the project.... more (see page 110)
Deleting Strings (see page 112)	Creating this VCL application consists of the following steps: 1. Create a VCL Form with Buttons and ListBox controls. 2. Write the code to add strings to a list. 3. Write the code to delete a string from the list. 4. Run the application.
Displaying an Auto-Created VCL Form (see page 113)	Using RAD Studio, the following procedure creates a modal form at design time that is displayed later during program execution. Building this VCL application consists of the following steps: 1. Create the project directory. 2. Create two forms for the project. 3. Link the forms. 4. Create a control on the main form to display the modal form; then write the event handler. 5. Build and run the application.

Displaying a Bitmap Image in a VCL Forms Application (see page 114)	This procedure loads a bitmap image from a file and displays it to a VCL form. 1. Create a VCL form with a button control. 2. Provide a bitmap image. 3. Code the button's onClick event handler to load and display a bitmap image. 4. Build and run the application.
Displaying a Full View Bitmap Image in a VCL Forms Application (see page 116)	This procedure loads a bitmap image from a file and displays it in its entirety to a VCL form. The procedure uses the Height and Width properties of the Bitmap object to display a full view of the image. 1. Create a VCL form with a button control. 2. Provide a bitmap image. 3. Code the button's onClick event handler to load and display a bitmap image. 4. Build and run the application.
Drawing a Polygon in a VCL Forms Application (see page 117)	This procedure draws a polygon in a VCL form. 1. Create a VCL form. 2. Code the form's OnPaint event handler to draw a polygon. 3. Build and run the application.
Drawing Rectangles and Ellipses in a VCL Forms Application (see page 118)	This procedure draws a rectangle and ellipse in a VCL form. 1. Create a VCL form. 2. Code the form's OnPaint event handler to draw a rectangle and ellipse. 3. Build and run the application.
Drawing a Rounded Rectangle in a VCL Forms Application (see page 118)	This procedure draws a rounded rectangle in a VCL form. 1. Create a VCL form and code the form's OnPaint event handler. 2. Build and run the application.
Drawing Straight Lines In a VCL Forms Application (see page 119)	This procedure draws two diagonal straight lines on an image in a VCL form. 1. Create a VCL form. 2. Code the form's OnPaint event handler to draw the straight lines. 3. Build and run the application.
Dynamically Creating a VCL Modal Form (see page 120)	You may not want all your VCL application's forms in memory at once. To reduce the amount of memory required at load time, your application can create forms only when it needs to make them available for use. A dialog box, for example, needs to be in memory only during the time the user interacts with it. Using RAD Studio, the following procedure creates a modal form dynamically. The main difference between dynamically creating a form and displaying an auto-created VCL form is that you remove the second form's invocation at startup and write code to dynamically create the form.... more (see page 120)

Dynamically Creating a VCL Modeless Form (see page 121)	A modless form is a window that is displayed until it is either obscured by another window or until it is closed or minimuzed by the user. Using RAD Studio, the following procedure creates a modeless form dynamically. Building this VCL application consists of the following steps: 1. Create the project directory. 2. Create two forms for the project. 3. Remove the second form's invocation at startup. 4. Link the forms. 5. Create a control on the main form to create and display the modal form; then write the event handler. 6. Build and run the application.
Iterating Through Strings in a List (see page 123)	This VCL application first creates a list of strings. Then it iterates through the strings, changing all string characters to uppercase. It consists of the following steps: 1. Create a VCL Form with Buttons and TListBox controls. 2. Write the code to create a string list and add strings to it. 3. Write the code to iterate through the string list to process string characters. 4. Run the application.
Building a Multithreaded Application (see page 124)	These are the essential steps to building a VCL Forms multithreaded application with a thread object using RAD Studio.
Writing Cleanup Code (see page 125)	
Avoiding Simultaneous Thread Access to the Same Memory (see page 125)	Use these basic techniques to prevent other threads from accessing the same memory as your thread: 1. Lock objects. 2. Use critical sections. 3. Use a multi-read exclusive-write synchronizer
Defining the Thread Object (see page 126)	
Handling Exceptions (see page 129)	
Initializing a Thread (see page 130)	
Using the Main VCL Thread (see page 131)	Using the main VCL thread consists of the following basic steps: 1. Create a separate routine to handle Windows messages received by components in your application. 2. Call CheckSynchronize periodically. 3. Declare thread-local variables, as necessary, for exclusive use by your thread.
Waiting for Threads (see page 132)	The following are procedures that can be used to wait for threads. • Wait for a thread to finish executing. • Wait for a task to complete. • Check if another thread is waiting for your thread to terminate.
Writing the Thread Function (Procedure) (see page 134)	The Execute method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program, because you must make sure that you do not overwrite memory that is used by other processes in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

Placing A Bitmap Image in a Control in a VCL Forms Application (see page 135)	This procedure adds a bitmap image to a combo box in a VCL forms application. 1. Create a VCL form. 2. Place components on the form. 3. Set component properties in the Object Inspector. 4. Write event handlers for the component's drawing action. 5. Build and run the application.
Reading a String and Writing It To a File (see page 136)	Creating this VCL application consists of the following steps: 1. Create a VCL Form with a button control. 2. Write the code to read the string and write it to a file. 3. Run the application.
Renaming Files (see page 137)	Creating this VCL application consists of the following steps: 1. Create a project directory containing a file to rename. 2. Create a VCL Form with button and label controls. 3. Write the code to rename the file. 4. Run the application.
Adding and Sorting Strings (see page 138)	Creating this VCL application consists of the following steps: 1. Create a VCL Form with Button, Label, and TListBox controls. 2. Write the code to add and sort strings. 3. Run the application.
Creating a VCL Forms ActiveX Button (see page 139)	Like a Delphi control, an ActiveX control generates program code when you place the component on a form or other logical container in the IDE. The main difference between an ActiveX control and a Delphi control is that an ActiveX control is language independent. You can create ActiveX controls for deployment to a variety of programming environments on Windows, not just Delphi or C++Builder. This procedure uses the VCL forms ActiveX wizard to create an ActiveX control. To test the control, you can install it on your machine as a VCL component in the IDE. To install the control, you... more (see page 139)
Creating a VCL Forms ActiveX Active Form (see page 140)	Like a Delphi control, an ActiveX control generates program code when you place the component on a form or other logical container in the IDE. The main difference between an ActiveX control and a Delphi control is that an ActiveX control is language independent. You can create ActiveX controls for deployment to a variety of programming environments on Windows, not just Delphi or C++Builder, for example. This procedure uses the VCL forms ActiveX Active Form wizard to create an Active Form containing two components. To test the control, you can deploy it to the Web. This procedure consists of the... more (see page 140)
Building a VCL Forms Web Browser Application (see page 142)	Creating the Web browser application consists of the following steps: 1. Create a VCL Form with a button control. 2. Add a TWebBrowser component to the form. 3. Add controls to enter a URL and launch the browser. 4. Write the code to launch the browser when a button is clicked. 5. Run the application.

Creating an Application that Uses Ribbon Controls (see page 143)	This procedure describes how to create an application that uses ribbon controls. The core ribbon functionality is derived from the TRibbon component. While the ribbon uses other components, none of the core components are registered on the tool palette. **Components:** 1. **TRibbon:** Main visual component that provides most functionality. 2. **TRibbonApplicationMenuBar:** Component that provides the functionality of the application menu. 3. **TRibbonQuickAccessToolbar:** Component that provides the functionality of the Quick Access Toolbar 4. **TRibbonPage:** Component that represents the page of the ribbon that is currently visible 5. **TRibbonGroup:** Component that all of the pages commands are displayed in. Commands must be placed in a group.... more (see page 143)
Adding Commands to the Ribbon (see page 144)	This topic follows in sequence the creation of a ribbon application using either the Ribbon Application Wizard or the manual steps described in Creating an Application that Uses Ribbon Controls (see page 143). This topic assumes that you are familiar with the TActionManager component and the components associated with its use. Numerous new properties have been added to help support the ribbon requirements. Many existing properties have no effect when modified on a command that is displayed on the ribbon. For instance: • Small buttons always display their glyph to the left of the caption. • Large buttons always display their glyph at... more (see page 144)

2.5.1 Building a Windows "Hello World" Console Application

This "Hello World" console application demonstrates the essential steps for creating a Windows application in Delphi or C++. The application uses Windows, a console window, an event, and will display a dialog in response to a user action.

To create the "Hello world" console application

1. Create a Windows console application.
2. Create the logic.
3. Run the application.

To create a Windows console application

1. Choose **File** ▶ **New** ▶ **Other...**. The **New Items** dialog box appears.
2. In the **New Items** dialog box, select either **Delphi Projects** or **C++Builder Projects** and then double-click **Console Application**.
3. For C++, in the **New Console Application** dialog box, make sure that **VCL Forms** and **Multi-threaded** are unchecked, and **Console application** is checked. Then click **OK**.

To associate code with the console window

1. In the code template that is displayed in the **Code Editor**: For Delphi, enter the following statements after the **try** keyword and before the **except** keyword:

```
WriteLn('Hello, World!');

  ReadLn;
```

For C++, enter the following after `#pragma hdrstop`:

```
#include <iostream>
```

2. For C++, enter the following code after the opening brace ({):

```
std::cout<<"Hello, World!"<<std::endl;
std::cin.ignore();
```

3. Save the application.

To run the "Hello World" application

1. Choose **Run** ▶ **Run**. The application compiles and displays a console window with your "Hello World" message.
2. Press the ENTER key. The console window closes when the program terminates.

See Also

Windows Overview (see page 45)

Building a Windows Application (see page 89)

Building Applications Menus (see page 90)

2.5.2 Developing a Windows Application

The following procedure illustrates the essential steps for building a Windows application.

To create a Windows project

1. Select **New** ▶ **VCL Forms Application — Delphi for Win32**. The **Windows Designer** displays.
2. If necessary, select **Design** view.
3. From the **Tool Palette**, drag components onto the designer to create the user interface.
4. Associate logic with the controls.

To associate code with a control

1. In the Designer, double-click the component to which you want to apply logic. The **Code Editor** appears, cursor in place between the reserved words **begin** and **end** in the event handler.
2. Code your application logic.
3. Save and compile the application.

See Also

Windows Overview (see page 45)

Building a Windows "Hello World" Application (see page 88)

Building Windows Application Menus (see page 90)

2.5.3 Building Application Menus

Menus provide an easy way for your users to execute logically grouped commands. You can add or delete menu items, or drag them to rearrange them during designtime. In addition to **TMainMenu** and **TPopupMenu** components, the **Tool Palette** also contains **TActionMainMenuBar**, **TActionManager**, and **TActionToolBar**.

To create application menus

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **VCL Forms Application.** The **Windows Designer** displays.
3. Build application menus.
4. Use the Menu Designer.
5. Create an event handler for each menu item.
6. Move menu items.
7. Add images to menu items.

To build application menus

1. From the **Standard** category of the **Tool Palette**, add **TMainMenu** or **TPopupMenu** component to your form. A visual representation of the menu appears on the designer.

   Note: A TMainMenu

   component creates a menu that is attached to the title bar of the form. A **TPopupMenu** component creates a menu that appears when the user right-clicks in the form.
2. To view the menu, if the form is visible, click the form. The menu appears in the form exactly as it will when you run the program.
3. To delete a menu item, select the menu item you want to delete. Press `Delete`.
4. To edit menu items, select the VCL form, select the menu item you want to edit, and edit its properties.
5. To make the menu item a separator bar, place the cursor on the menu where you want a separator to appear and enter a hyphen (-) for the caption or press the hyphen (-) key.
6. To specify a keyboard shortcut for a menu item, in the **Object Inspector**, set the ShortCut property.

To use the Menu Designer

1. Select a menu component on the form.
2. Double-click the menu component. The **Menu Designer** window opens.

   Note: You can also open the Menu Designer

   by clicking the ellipsis(...) button next to the Items property in the **Object Inspector**.
3. To name a menu component, in the **Object Inspector**, set the Caption property.

   Tip: Delphi derives the Name property from the caption, for e.g. if you give a menu item a Caption property value of File, Delphi assigns the menu item a Name property of File1. However, if you fill in the Name property before filling in the Caption property, Delphi leaves the Caption property blank until you type a value.
4. Right-click anywhere on the Menu Designer to use the **Menu Designer** context menu. A drop-down list opens. This is the **Menu Designer** context menu.
5. To insert a placeholder below or to the right of the cursor, choose **Insert** from the context menu.

6. To delete the selected menu item (and all its subitems, if any), click **Delete** from the context menu.
7. To switch among menus in a form, choose **Select Menu** from the context menu. The **Select Menu** dialog box appears. It lists all the menus associated with the form whose menu is currently open in the Menu designer.
8. From the list in the **Select Menu** dialog box, choose the menu you want to view or edit.

To create an event handler for a menu item

1. In the designer, double-click the menu item to which you wish to add an event handler. The **Code Designer** appears, cursor in place between event handler brackets.
2. Code your menu item logic.
3. Save and compile the application.

To move menu items

1. To move a menu item along the desired menu bar, drag the item until the arrow tip of the cursor points to the new location.
2. Release the mouse button.
3. To move a menu item into a menu list, drag the item until the arrow tip of the cursor points to the new menu.
4. Release the mouse button.

To add images to menu items

1. From the **Tool Palette**, drag a **TMainMenu** or **TPopupMenu** component to a form.
2. From the **Tool Palette**, drop a **TImageList** component to the form.
3. Double-click on the **TImageList** component. The **ImageList** editor opens.
4. Click **Add** to select the bitmap or bitmap group you want to use in the menu.
5. Select the bitmap that you want and click **OK**.
6. In the **Object Inspector**, set the **Images** property of the **TMainMenu** or **TPopupMenu** component to the image you selected in the **ImageList** editor.

See Also

Windows Overview (see page 45)

Building a Windows Application (see page 89)

2.5.4 Building a VCL Forms Application with Decision Support Components

Creating a form with tables and graphs of multidimensional data consists of the following major steps:

1. Create a VCL form.
2. Add a decision query and dataset.
3. Add a decision cube.
4. Add a decision source.
5. Optionally add a decision pivot.
6. Add one or more decision grids and graphs.
7. Set the active property of the decision query (or alternate dataset component) to True.

To create a VCL form

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** displays.
2. If necessary, click Form1 to make it the active window.

To add a decision dataset

1. From the **Decision Cube** page on the **Tool Palette**, add a DecisionQuery component to the top of the form.

   Tip: Place non-visual components such as this one in the top left corner of the form to keep them out of the way of visual components you will be adding.

2. Right-click the DecisionQuery component, and select **Decision Query Editor...**. The **Decision Query Editor** displays.
3. On the **Dimensions/Summary** tab, select the BCDEMOS database from the **Database:** drop-down list.
4. From the **Table:** drop-down, select the parts.db table. The **List of Available Fields:** listbox displays the fields in the parts.db table.
5. Use CTRL+Click to select the PartNo, OnOrder, and Cost fields; then click the right-arrow button next to the **Dimensions:** listbox. PartNo, OnOrder, and Cost display in the listbox.
6. Select the OnOrder field; then click the right-arrow button next to the **Summaries:** listbox and select count from the pop-up that displays. COUNT(OnOrder) displays in the **Summaries:** listbox.
7. Select the Cost field in the **List of Available Fields:** listbox; then click the right-arrow button next to the **Summaries:** listbox and select sum from the pop-up that displays. SUM(Cost) displays in the **Summaries:** listbox.
8. Click **OK** to close the **Decision Query Editor**.

   Note: When you use the Decision Query Editor, the query is initially handled in ANSI-92 SQL syntax and then translated (if necessary) into the dialect used by the server. The Decision Query editor reads and displays only ANSI standard SQL. The dialect translation is automatically assigned to the TDecisionQuery's SQL property. To modify a query, edit the ANSI-92 version in the Decision Query rather than the SQL property.

To add a decision cube

1. From the **Decision Cube** page on the **Tool Palette**, add a decision cube component to the top left corner of the form.
2. In the **Object Inspector**, select DecisionQuery1 from the drop-down list next to the decision cube's DataSet property.

To add a decision source

1. From the **Decision Cube** page on the **Tool Palette**, add a decision source component to the top left corner of the form.
2. In the **Object Inspector**, select DecisionCube1 from the drop-down list next to the decision source's DecisionCube property.

To add a decision pivot

1. From the **Decision Cube** page on the **Tool Palette**, add an optional DecisionPivot component to the top of the form.

   Tip: The decision pivot displays in the final application window. Place it to the right of the nonvisual components.

2. In the **Object Inspector**, select DecisionSource1 from the drop-down list next to the decision pivot's DecisionSource property.

To create a decision grid

1. From the **Decision Cube** page on the **Tool Palette**, add a decision grid component to the form just beneath the decision pivot.
2. In the **Object Inspector**, select DecisionSource1 from the drop-down list next to the decision grid's DecisionSource property.

To create a decision graph

1. From the **Decision Cube** page on the **Tool Palette**, add a decision graph component to the form just beneath the decision grid.
2. In the **Object Inspector**, select DecisionSource1 from the drop-down list next to the decision graph's DecisionSource property.

To run the application

1. In the **Object Inspector**, select True from the Active property drop-down. The visual decision graph, grid, and pivot components display data.
2. Choose **Run** ▶ **Run** to run the application. The application runs and displays the decision support components.
3. Use the decision pivot to update, as desired, the data displayed in the grid and graph.

See Also

VCL Overview (see page 36)

2.5.5 Building VCL Forms Applications With Graphics

Each of the procedures listed below builds a VCL Form application that uses graphics. Build one or more of the examples and then add other graphics features to these basic VCL Form applications.

1. Draw straight lines.
2. Draw rectangles and ellipses.
3. Draw a polygon.
4. Display a bitmap image.
5. Place a bitmap in a combo box.

See Also

Overview of Graphics Programming (see page 2176)

Drawing Straight Lines In a VCL Application (see page 119)

Drawing Rectangles and Ellipses in a VCL Application (see page 118)

Displaying a Bitmap Image in a VCL Application (see page 114)

Placing A Bitmap Image In a Combo Box of a VCL Application (see page 135)

2.5.6 Building a VCL Forms MDI Application Using a Wizard

The VCL Forms MDI application wizard automatically creates a project that includes the basic files for an MDI application. In addition to the Main source file, the wizard creates unit files for child and about box windows, along with the supporting forms files and resources.

To create a new MDI application using a wizard

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** and double-click the **MDI Application** icon. The **Browse to Folder** dialog box is displayed.

2. Navigate to the folder in which you want to store the files for the project.
3. Click **OK**.
4. Choose **Run** ▶ **Run** to compile and run the application.
5. Try commands that are automatically set up by the MDI Application wizard.

See Also

VCL Overview (see page 36)

2.5.7 Building a VCL Forms MDI Application Without Using a Wizard

The basic steps to create a new MDI application with a child window without using a wizard are

1. Create a main window form (MDI parent window).
2. Create a child window form.
3. Have the main window create the child window under user control.
4. Write the event handler code to close the child window.
5. Create the main menu and commands.
6. Add the event handlers for the commands.
7. Compile and run the application.

To create the main window form

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. In the **Object Inspector**, set the FormStyle property to fsMDIForm.
3. Enter a more descriptive name such as `frMain` for the Name property.
4. Save the unit file with a more descriptive name, such as `uMain.pas` (Delphi) or `uMain.cpp` (C++).

To create a child window

1. Choose **File** ▶ **New** ▶ **Form**
2. In the **Object Inspector**, set the FormStyle property to fsMDIChild.
3. Enter a more descriptive name such as frChild for the Name property.
4. Save the unit file as `uChild.pas` (Delphi) or `uChild.cpp` (C++).

To have the main window create the child window

1. Choose **Project** ▶ **Options** ▶ **Forms**. The **Project Options** dialog box appears.
2. Select **frChild** from **Auto-create forms** list and click **[>]** to move it to the **Available forms** list and click **OK**.
3. Select the frMain form to activate it; then switch to the Code view.
4. For Delphi, scroll to the `uses` section and add `uChild` to the list. For C++, add `#include "uChild.h"` to `uMain.h`.
5. For Delphi, scroll to the `private` declarations section and enter this procedure declaration:

```
procedure CreateChildForm(const childName: string);
```

For C++, add the following function declaration to the **private:** declarations of `TfrMain`.

```
void __fastcall CreateChildForm( const AnsiString childName );
```

6. For Delphi, scroll to the `implementation` section, and enter the code below:

```
procedure TfrMain.CreateChildForm (const childName: string);
  var Child: TfrChild;
  begin
    Child := TfrChild.Create(Application);
    Child.Caption := childName;
  end;
```

For C++, add the following function definition to `uMain.cpp`:

```
void __fastcall TfrMain::CreateChildForm( const AnsiString childName )
{
        TfrChild *Child = new TfrChild( Application );
        Child->Caption = childName;
}
```

To write the event handler code to close the child window

1. If necessary, activate the frMain form; then select the **Events** tab in the **Object Inspector**.
2. Double-click the OnClose event. The **Code Editor** displays with the cursor in the `TfrMain.FormClose` (Delphi) or `TfrMain::FormClose` (C++) event handler block.
3. For Delphi, enter the following code:

```
Action := caFree;
```

For C++, enter the following code:

```
Action = caFree;
```

To create the main menu and commands

1. From the **Standard** page on the Tool Palette, place a TMainMenu component on the main form.
2. Double-click the TMainMenu component. The Menu designer (frMain.MainMenu1) displays with the first blank menu item highlighted.
3. In the **Object Inspector** on the **Properties** tab, enter mnFile for the Name property and &File for the Caption property; then press ENTER. In the Menu designer, File displays as the name of the first menu item.
4. In the Menu designer, select File. A blank command field displays in the File group. Select the blank command.
5. In the **Object Inspector**, enter mnNewChild for the Name property and &New child for the Caption property; then press ENTER. In the Menu designer, New child displays as the name of the first file command, and a blank command field displays just beneath New child.
6. Select the blank command.
7. In the **Object Inspector**, enter mnCloseAll for the Name property and &Close All for the Caption property; then press ENTER. In the Menu designer, Close All displays as the name of the second file command.

To add event handlers for the New child and Close All commands

1. If necessary, open the Menu designer and select New child.
2. In the **Object Inspector**, double-click the OnClick event on the **Events** tab. The **Code Editor** displays with the cursor in the `TfrMain.mnNewChildClick` (Delphi) or `TfrMain::mnNewChildClick` (C++) event handler block.
3. For Delphi, enter the following code:

```
CreateChildForm('Child '+IntToStr(MDIChildCount+1));
```

For C++, enter the following code:

```
CreateChildForm( “Child “ + IntToStr( MDIChildCount + 1 ) );
```

4. In the Menu designer, select Close All.

5. In the **Object Inspector**, double-click the OnClick event on the **Events** tab. The **Code Editor** displays with the cursor in the `TfrMain.mnCloseAllClick` (Delphi) or `TfrMain::mnCloseAllClick` (C++) event handler block.
6. For Delphi, enter the following code:

```
for i:=0 to MDIChildCount - 1 do
  MDIChildren[i].Close;
```

For C++, enter the following code:

```
for( int i = 0; i < MDIChildCount; i++ ) {
        MDIChildren[i]->Close();
}
```

7. For Delphi, declare the local variable `i`. The first two lines of the event handler code should appear as shown here when you are done:

```
procedure TfrMain.mnCloseAllClick(Sender: TObject);
  var i: integer;
```

Note: The event handler minimizes the child window in the main window. To close the child window, you must add an OnClose procedure to the child form (next).

To close the child window

1. Activate the child form.
2. In the **Object Inspector**, double-click the OnClose event on the **Events** tab. The **Code Editor** displays with the cursor in the `TfrChild.FormClose` (Delphi) or `TfrChild::FormClose` (C++) event handler block.
3. For Delphi, enter the following statement:

```
Action := caFree;
```

For C++, enter the following statement:

```
Action = caFree;
```

To compile and run the MDI application

1. Choose **Run** ▸ **Run**.
2. The application executes, displaying the File command.
3. Choose **File** ▸ **New child** one or more times. A child window displays with each New child command.
4. Choose **File** ▸ **Close All**. The child windows close.

See Also

VCL Overview (see page 36)

2.5.8 Building a VCL Forms SDI Application

To create a new SDI application

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** and double-click the **SDI Application** icon.
2. Pick a folder to save the files in and click **OK**.
3. Choose **Run** ▸ **Run** to compile and run the application.

See Also

VCL Overview (see page 36)

2.5.9 Creating a New VCL Component

You can use the **New VCL Component** wizard to create a new VCL component for your application. The wizard detects the personality of the product you are using and creates the appropriate type of component.

Metaprocedure: Creating a new VCL component

1. Specify an ancestor component on the **Ancestor Component** page.
2. Specify the class name and other properties on the **Component** page.
3. Choose from three methods for creating a unit:

- Create a unit, on the **Create unit** page
- Install the unit to an existing package, on the **Existing package** page
- Install the unit as a new package, on the **New Package** page

To specify an ancestor component

1. Create a new package for the new component by choosing **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** and double-clicking the **Package** icon. This step adds a new package to the **Project Manager** and enables the **Add unit to <my_new_package>** option on the **Create unit** page.
2. In the IDE, choose **Component** ▸ **New VCL Component**. This step displays the first page of the **New VCL Component** wizard (the **Ancestor Component** page) and loads the page with names of components, along with the unit that defines each component.
3. On the **Ancestor Component** page, select an ancestor component from the list.
4. Click **Next**. This displays the **Component** page of the **New VCL Component** wizard.

To specify the class name and other properties

1. On the **Component** page, the fields are prepopulated for **Class Name**, **Palette Page**, **Unit name**, and **Search path**. You can either accept the default values for these fields, or you can change the values by using the following steps. If you want to accept the default values, skip to step 6.
2. To change the default class name, enter a different class name in the **Class Name** textbox.
3. To change the default palette page for the component, in the **Palette Page** textbox, either enter the name of your chosen **Tool Palette** area, or click the down-arrow to select from palette page areas.
4. In the **Unit Name** textbox, you can either enter or edit the full path of the unit name. Click [...] to browse for and select the unit you want.
5. Either enter or edit the search path in the **Search Path** textbox.
6. Click **Next**. This opens the **Create unit** page of the wizard. The choices on the **Create unit** page are:

- **Create Unit**
- **Install to Existing Package**
- **Install to New Package**
- **Add unit to <my_new_package>** (this selection is present only if you create a new package in the very first step, above)

To create a unit

1. On the **Create Unit** page, select the **Create Unit** radio button.
2. Click **Finish**. The new unit opens in the **Code Editor**.

To install a unit into an existing package

1. On the **Create unit** page, select the **Install to Existing Package** radio button.
2. Click **Next**. This generates a list of existing packages.
3. On the **Existing Package** page, select the package you want to install the unit into.
4. Click **Finish**. The new unit opens in the **Code Editor**.

To install a unit into a new package

1. On the **Create unit** page, select the **Install to New Package** radio button.
2. Click **Next**.
3. On the **New Package** page, enter a name for the package into the **File Name** textbox.
4. In the **Description** textbox, enter a description of your new package .
5. Click **Finish**. The new unit opens in the **Code Editor**.

To install a unit into the package created before starting the wizard

1. On the **Create Unit** page, select **Add unit to <my_new_package>.**
2. Click **Finish**. The new unit opens in the **Code Editor**.

See Also

VCL Overview (see page 36)

2.5.10 Building a VCL Forms ADO Database Application

The following procedure describes how to build an ADO database application.

Building a VCL ADO application consists of the following major steps:

1. Set up the database connection.
2. Set up the dataset.
3. Set up the data provider, client dataset, and data source.
4. Connect a DataGrid to the connection components.
5. Run the application.

To add an ADO connection component

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **dbGo** page of the **Tool Palette**, place an ADOConnection component on the form.
3. Double-click the ADOConnection component to display the **ConnectionString** dialog.
4. If necessary, select **Use Connection String**; then click the **Build** button to display the **Data Link Properties** dialog.
5. On the **Provider** page of the dialog, select **Microsoft Jet 4.0 OLE DB Provider**; then click the **Next** button to display the **Connection** page.
6. On the **Connection** page, click the ellipsis button to browse for the **dbdemos.mdb** database. The default path to this database is C:\Program Files\Common Files\CodeGear Shared\Data.
7. Click **Test Connection** to confirm the connection. A dialog appears, indicating the status of the connection.

8. Click **OK** to close the **Data Link Properties** dialog. Click **OK** to close the **ConnectionString** dialog.

To set up the dataset

1. From the **dbGo** page, place an ADODataSet component at the top of the form.
2. In the **Object Inspector**, select the Connection property drop-down list. Set it to ADOConnection1.
3. Set the CommandText property to an SQL command, for example, Select * from orders. You can either type the Select statement in the **Object Inspector** or click the ellipsis to the right of CommandText to display the **CommandText Editor** where you can build your own query statement.

   Tip: If you need additional help while using the CommandText Editor

   , click the **Help** button.
4. Set the Active property to True to open the dataset. You are prompted to log in. Use admin for the username and no password.

To add the provider

1. From the **Data Access** page, place a DataSetProvider component at the top of the form.
2. In the **Object Inspector**, select the DataSet property drop-down list. Set it to ADODataSet1.

To add client dataset

1. From the **Data Access** page, place a ClientDataSet component to the right of the DataSetProvider component on the form.
2. In the **Object Inspector**, select the ProviderName drop-down. Set it to DataSetProvider1.
3. Set the Active property to True to allow data to be passed to your application. A data source connects the client dataset with data-aware controls. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component for their data to be displayed and manipulated in data-aware controls on the form.

To add the data source

1. In the **Tool Palette** on the **Data Access** page, place a DataSource component to the right of the ClientDataSet on the form.
2. In the **Object Inspector**, select the DataSet property drop-down. Set it to ClientDataSet1.

To connect a DataGrid to the DataSet

1. In the **Tool Palette** on the **Data Controls** page, place a DBGrid component on the form.
2. In the **Object Inspector**, select the DataSource property drop-down. Set the data source to DataSource1.
3. Choose **Run** ▶ **Run**.
4. You are prompted to log in. Enter admin for the username and no password. The application compiles and displays a VCL form with a DBGrid.

See Also

VCL Overview (see page 36)

2.5.11 Building a VCL Forms Application

The following procedure illustrates the essential steps to building a VCL Forms application using RAD Studio.

To create a VCL Form

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The

VCL Forms Designer is displayed.

2. From the **Tool Palette**, place components onto the form to create the user interface.
3. Write the code for the controls.

To associate code with a control

1. Double-click the component on the form to which you want to apply logic. The **Code Editor** displays, cursor in place within the event handler block.
2. Code your application logic.
3. Save and compile the application.

See Also

VCL Overview (see page 36)

Building a VCL Forms "Hello World" Application (see page 101)

Building a VCL Forms MDI Application Without Using a Wizard (see page 94)

Using ActionManager to Create Actions in a VCL Forms Application (see page 102)

2.5.12 Creating Actions in a VCL Forms Application

Using RAD Studio, the following procedure illustrates how to create actions using the ActionList tool. It sets up a simple application and describes how to create a file menu item with a file open action.

Building the VCL application with ActionList actions consists of the following major steps:

1. Create a main window and add tools for creating a main menu and a File open action.
2. Add the File category to the main menu.
3. Add the File open action to the File category.
4. Build and run the application.

To create a main window

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** displays.
2. From the **Standard** category of the **Tool Palette**, add a TMainMenu and TActionList component to the form.

   Tip: To display icons for nonvisual components such as ActionList1, choose Tools->Options->Environment Options , select VCL Designer from the Delphi options, click Show Component Options, and click **OK**.

To add the File category to the main menu

1. Double-click MainMenu1 on the form. The MainMenu1 editor displays with the first blank command category selected.
2. In the **Object Inspector**, enter &File for the Caption property and press RETURN. File displays on the main menu.
3. Click File on the MainMenu1 editor. The first blank action under the File command displays. Select the blank action.
4. Double-click ActionList1 on the form. The ActionList editor displays.
5. In the editor, select **New Standard Action** from the drop-down list to display the **Standard Action Classes** dialog.
6. Scroll to the File category, and click the TFileOpen action.
7. Click **OK** to close the dialog. File displays in the Categories listbox in the ActionList editor.

8. Click File in the editor. The FileOpen1 action displays in the Action listbox.

To add the File Open action to the File category

1. Double-click MainMenu1, if necessary, to display the MainMenu1 editor; select the blank action under the File category.
2. In the **Object Inspector**, enter &Open for the Caption property and select FileOpen1 from the Action property drop-down list; then press RETURN. Open... displays in the blank action field in the MainMenu1 editor.

To build and run the application

1. Choose **Run** ▶ **Run**. The application executes, displaying a form with the main menu bar and the File menu.
2. Choose **File** ▶ **Open** in the application. The standard **Open** file dialog displays.

See Also

VCL Overview (see page 36)

2.5.13 Building a VCL Forms "Hello World" Application

This procedure demonstrates how to construct a simple “Hello world” VCL Forms application using either Delphi or C++. Though simple, the Windows Forms "Hello world" application demonstrates the essential steps for creating a VCL Forms application. The application uses a VCL Form, a control, an event, and displays a dialog in response to a user action.

Creating the "Hello world" application consists of the following steps:

1. Create a VCL Form with a button control.
2. Write the code to display "Hello world" when the button is clicked.
3. Run the application.

To create a VCL Form

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or **C++Builder Projects** and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. Click the VCL form to display the form view.
3. From the **Standard** page of the Tool Palette, place a TButton component on the form.

To display the "Hello world" string

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action field on the **Events** tab. The **Code Editor** displays, with the cursor in the `Button1Click` event handler block.
3. For Delphi, move the cursor before the `begin` reserved word and then press ENTER. This creates a new line above the code block. For C++, place the cursor after the opening brace ({) and press ENTER.
4. For Delphi, insert the cursor on the new line created, and type the following variable declaration:

```
var s: string;
```

For C++, enter the following code:

```
AnsiString s;
```

5. For Delphi, insert the cursor within the code block and type the following code:

```
s:= 'Hello world!';
ShowMessage(s);
```

For C++, enter the following code:

```
s = “Hello world!”;
ShowMessage( s );
```

To run the "Hello world" application

1. Choose **Run** ▶ **Run** to build and run the application. The form displays with a button called **Button1**.
2. Click **Button1**. A dialog box displays the message "Hello World!"
3. Close the VCL form to return to the IDE.

See Also

VCL Overview (see page 36)

2.5.14 Using ActionManager to Create Actions in a VCL Forms Application

Using RAD Studio, the following procedure illustrates how to create actions using ActionManager. It sets up a simple user interface with a text area, as would be appropriate for a text editing application, and describes how to create a file menu item with a file open action.

Building the VCL application with ActionManager actions consists of the following major steps:

1. Create a main window.
2. Add a File open action to the ActionManager.
3. Create the main menu.
4. Add the action to the menu.
5. Build and run the application.

To create a main window and add a File open action

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Additional** page of the **Tool Palette**, add an TActionManager component to the form.
3. Double-click the TActionManager to display the Action Manager editor.

   Tip: To display captions for nonvisual components such as ActionManager, choose Tools->Environment Options . On the **Designer** tab, check **Show component captions**, and click **OK**.
4. If necessary, click the **Actions** tab.
5. Select **New Standard Action** from the drop-down list to display the **Standard Action Classes** dialog.
6. Scroll to the File category, and click the TFileOpen action.
7. Click **OK** to close the dialog.
8. In the Action Manager editor, select the File category. **Open...** displays in the **Actions:** list box.
9. Click **Close** to close the editor.

To create the main menu

1. From the **Additional** page of the **Tool Palette**, place an TActionMainMenuBar component on the form.
2. Open the Action Manager editor, and select the **File** category from the **Categories:** list box.
3. Drag File to the blank menu bar. File displays on the menu bar.

To build and run the application

1. Choose **Run** ▶ **Run**. The application executes, displaying a form with the main menu bar and the File menu.
2. Choose **File** ▶ **Open**. The Open file dialog displays.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

Building a VCL Forms MDI Application Without Using a Wizard (see page 94)

2.5.15 Building a VCL Forms dbExpress Database Application

The following procedure describes how to build a dbExpress database application.

Building a VCL Forms dbExpress application consists of the following major steps:

1. Set up the database connection.
2. Set up the unidirectional dataset.
3. Set up the data provider, client dataset, and data source.
4. Connect a DataGrid to the connection components.
5. Run the application.

To add a dbExpress connection component

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **dbExpress** page of the **Tool Palette**, place a TSQLConnection component on the form.
3. Double-click the TSQLConnection component to display the **Connection Editor**.
4. In the **Connection Editor**, set the **Connection Name** field to IBConnection.
5. In the **Connections Setting** box, specify the path to the InterBase database file called employee.gdb in the Database field. By default, the file is located in `C:\Program Files\Common Files\Borland Shared\Data`.
6. Accept the value in the **User_Name** field (`sysdba`) and **Password** field (`masterkey`).
7. Click **OK** to close the **Connection Editor** and save your changes.

To set up the unidirectional dataset

1. In the **Tool Palette** on the **dbExpress** page, place a TSQLDataSet component at the top of the form.
2. In the **Object Inspector**, select the SQLConnection property drop-down list. Set it to TSQLConnection1.
3. Set the CommandText property to an SQL command, for example, `Select * from SALES`. You are prompted to log in. Use the `masterkey` password. For the SQL command, you can either type a Select statement in the Object Inspector or click the ellipsis to the right of CommandText to display the **CommandText Editor** where you can build your own query statement.

   Tip: If you need additional help while using the CommandText Editor

   , click the **Help** button.
4. In the **Object Inspector**, set the Active property to True to open the dataset.

To add the provider

1. In the **Tool Palette** on the **Data Access** page, place a TDataSetProvider component at the top of the form.
2. In the **Object Inspector**, select the DataSet property drop-down list. Set it to SQLDataSet1.

To add client dataset

1. In the **Tool Palette** on the **Data Access** page, place a TClientDataSet component to the right of the TDataSetProvider component on the form.
2. In the **Object Inspector**, select the ProviderName drop-down. Set it to DataSetProvider1.
3. Set the Active property to True to allow data to be passed to your application. A data source connects the client dataset with data-aware controls. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component for their data to be displayed and manipulated in data-aware controls on the form.

To add the data source

1. In the **Tool Palette** on the **Data Access** page, place a TDataSource component to the right of the TClientDataSet on the form.
2. In the **Object Inspector**, select the DataSet property drop-down. Set it to ClientDataSet1.

To connect a DataGrid to the DataSet

1. In the **Tool Palette** on the **Data Controls** page, place a TDBGrid component on the form.
2. In the **Object Inspector**, select the DataSource property drop-down. Set the data source to DataSource1.
3. Choose **Run** ▶ **Run**. You are prompted to enter a password. Enter `masterkey`. If you enter an incorrect password or no password, the debugger throws an exception. The application compiles and displays a VCL form with a DBGrid.

See Also

VCL Overview (see page 36)

2.5.16 Building an Application with XML Components

This example creates a VCL Forms application that uses an XMLDocument component to display contents in an XML file.

The basic steps are:

1. Create an XML document.
2. Create a VCL form.
3. Place an XMLDocument component on the form, and associate it with the XML file.
4. Create VCL components to enable the display of XML file contents.
5. Write event handlers to display XML child node contents.
6. Compile and run the application.

To create the XML document

1. Copy the text below into a file in a text editor.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings [
    <!ELEMENT StockHoldings (Stock+)>
    <!ELEMENT Stock (name)>
```

```
    <!ELEMENT Stock (price)>
    <!ELEMENT Stock (symbol)>
    <!ELEMENT Stock (shares)>
]>

<StockHoldings>
  <Stock exchange="NASDAQ">
    <name>CodeGear</name>
    <price>10.375</price>
    <symbol>BORL</symbol>
    <shares>100</shares>
  </Stock>

  <Stock exchange="NYSE">
    <name>MyCompany</name>
    <price>8.75</price>
    <symbol>MYCO</symbol>
    <shares type="preferred">25</shares>
  </Stock>
</StockHoldings>
```

2. Save the file to your local drive as an XML document. Give it a name such as `stock.xml`.
3. Open the document in your browser. The contents should display without error.

   Note: In the browser, you can choose View->Source

   to view the source file in the text editor.

To create a form with an XMLDocument component

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Internet** page on the **Tool Palette**, place an TXMLDocument component on the form.
3. In the **Object Inspector**, click the ellipsis button next to the FileName property, browse to the location of the XML file you created, and open it. The XML file is associated with the TXMLDocument component.
4. In the **Object Inspector**, set the Active property to true.

To set up the VCL components

1. From the **Standard** page on the **Tool Palette**, place a TMemo component on the form.
2. From the **Standard** page on the **Tool Palette**, place two TButton components on the form just above **Memo1**.
3. In the **Object Inspector** with **Button1** selected, enter `CodeGear` for the Caption property.
4. In the **Object Inspector** with **Button2** selected, enter `MyCompany` for the Caption property.

To display child node contents in the XML file

1. In the **Object Inspector** with **Button1** selected, double-click the OnClick event on the **Events** tab. The Code displays with the cursor in the `TForm1.Button1Click` event handler block.
2. Enter the following code to display the stock price for the first child node when the CodeGear button is clicked:

```
  BorlandStock:=XMLDocument1.DocumentElement.ChildNodes[0];
  Price:= BorlandStock.ChildNodes['price'].Text;
  Memo1.Text := Price;
IXMLNode *BorlandStock = XMLDocument1->DocumentElement->
            ChildNodes->GetNode(0);
WideString Price = BorlandStock->ChildNodes->
            FindNode( "price" )->Text;
Memo1->Text = Price;
```

3. For Delphi, add a `var` section just above the code block in the event handler, and enter the following local variable declarations:

```
var
     BorlandStock: IXMLNode;
  Price: string;
```

4. In the **Object Inspector** with **Button2** selected, double-click the OnClick event on the **Events** tab. The Code displays with the cursor in the `TForm1.Button2Click` event handler block.
5. Enter the following code to display the stock price for the second child node when the MyCompany button is clicked:

```
  MyCompany:=XMLDocument1.DocumentElement.ChildNodes[1];
  Price:= MyCompany.ChildNodes['price'].Text;
  Memo1.Text := Price;
IXMLNode *MyCompany = XMLDocument1->DocumentElement
       ->ChildNodes->GetNode(1);
WideString Price = BorlandStock->ChildNodes
       ->FindNode( "price" )->Text;
Memo1->Text = Price;
```

6. For Delphi, add a `var` section just above the code block in the event handler, and enter the following local variable declarations:

```
var
  MyCompany: IXMLNode;
  Price: string;
```

To compile and run the application

1. Choose **Run** ▶ **Run** to compile and execute the application. The application form displays two buttons and a memo.
2. Click the **CodeGear** button. The stock price displays.
3. Click the **MyCompany** button. The stock price displays.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

2.5.17 Copying Data From One Stream To Another

Creating this VCL application consists of the following steps:

1. Create a project directory containing a text file to copy.
2. Create a VCL Form with a button control.
3. Write the code to read the string and write it to a file.
4. Run the application.

To set up your project directory and a text file to copy

1. Create a directory in which to store your project files.
2. Using a text editor, create a simple text file and save it as `from.txt` in your project directory.

To create a VCL Form with a button control

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place a TButton component on the form.
3. In the **Object Inspector**, enter CopyFile for the Caption and Name properties.

To write the copy stream procedure

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.CopyFileClick` (Delphi) or `TForm1::CopyFileClick` (C++) event handler block.
3. For Delphi, Place the cursor before the `begin` reserved word; then press return. This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declaration:

```
var stream1, stream2: TStream;
```

For C++, enter the following variable declarations:

```
TStream *stream1, *stream2;
```

5. Insert the cursor within the code block, and type the following code:

```
stream1 := TFileStream.Create('from.txt', fmOpenRead);
try
  stream2:= TFileStream.Create('to.txt', fmCreate);
  try
    stream2.CopyFrom(stream1, stream1.Size);
  finally
    stream2.Free;
  end;
finally
  stream1.Free;
end;
stream1 = new TFileStream( “from.txt”, fmOpenRead );
try {
 stream2 = new TFileStream( “to.txt”, fmCreate );
     try {
   stream2->CopyFrom( stream1, stream1->Size );
 } __finally {
   stream2->Free();
 }
} finally {
 stream1->Free();
}
```

To run the application

1. Save your project files; then choose **Run** ▶ **Run** to build and run the application. The form displays with a button called **CopyFile**.
2. Click **CopyFile**.
3. Use a text editor to open the newly created file to.txt, which is located in your project directory. The contents of from.txt are copied into to.txt.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

2.5.18 Copying a Complete String List (VCL)

Copying a string list can have the effect of appending to or overwriting an existing string list. This VCL application appends to a string list. With a simple change, it can overwrite a string list. Creating this VCL application consists of the following steps:

1. Create a VCL Form with TButtons, TComboBox, and TMemo controls.
2. Write the code to create a string list to the Button1 OnClick handler.
3. Write the code to copy the string list to the Button2 OnClick handler.
4. Run the application.

To create a VCL Form with Button, ComboBox, and Memo controls

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place two TButtons, a TComboBox, and a TMemo component on the form.

To create the string list

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press `return`. This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declarations:

```
var
  StringList: TStrings;
```

For C++, enter the following variable declarations:

```
TStrings *StringList;
```

5. Insert the cursor within the code block, and type the following code:

```
StringList := TStringList.Create;
  try
    with StringList do begin
      Add('This example uses a string List.');
      Add('It is the easiest way to add strings');
      Add('to a combobox''s list of strings.');
      Add('Always remember: the TStrings.Create');
      Add('method is abstract; use the');
      Add('TStringList.Create method instead.');
    end;

    with ComboBox1 do begin
                        Width := 210;
      Items.Assign(StringList);
      ItemIndex := 0;
    end;
  finally
    StringList.free;
  end;
StringList = new TStringList();
try {
 StringList->Add( "This example uses a string list" );
 StringList->Add( "It is the easiest way to add strings" );
 StringList->Add( "to a ComboBox's list of strings." );
 StringList->Add( "Remember to call the TStringList constructor!" );
 ComboBox1->Width = 210;
 ComboBox1->Items->Assign( StringList );
 ComboBox1->ItemIndex = 0;
} __finally {
 StringList->Free();
}
```

To copy the string list

1. Select Button2 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.Button2Click` (Delphi) or `TForm1::Button2Click` (C++) event handler block. At the cursor, enter the following code:

```
Memo1.Lines.AddStrings(ComboBox1.Items);
Memo1->Lines->AddStrings( ComboBox1->Items
```

To run the application

1. Save your project files; then choose **Run** ▶ **Run** to build and run the application. The form displays with the controls.
2. Click Button1.
3. In ComboBox1, click the arrow to expand the drop-down list. The strings display in the TComboBox in the order listed in the event handler code for Button1.
4. Click Button2. In the Memo1 window, the strings from ComboBox1 are appended to the 'Memo1' string.

   Note: Try replacing the code in the Button2 event handler with the following code; then compile and run the application again.

```
Memo1.Lines.Assign(ComboBox1.Items);
Memo1->Lines->Assign( ComboBox1->Items );
```

The strings from ComboBox1 overwrite the 'Memo1' string.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

2.5.19 Creating Strings

Creating this VCL application consists of the following steps:

1. Create a VCL Form with TButton and TComboBox controls.
2. Write the code to create strings to the TButton OnClick handler.
3. Run the application.

To create a VCL Form with TButton and TComboBox controls

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place a TButton, a TLabel, and a TComboBox component on the form.

To write the create string procedure

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press `RETURN`. This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declarations:

```
var
  StringList: TStrings;
```

For C++, enter the following variable declarations:

```
TStrings *StringList;
```

5. Insert the cursor within the code block, and type the following code:

```
StringList := TStringList.Create;
  try
    with StringList do begin
      Add('Animals');
      Add('Cats');
      Add('Flowers');
    end;

    with ComboBox1 do begin
                        Width := 210;
      Items.Assign(StringList);
      ItemIndex := 0;
    end;

    Label1.Caption := 'Flowers has an index of ' +
      IntToStr( StringList->IndexOf( 'Flowers' ) );
  finally
    StringList.free;
  end;
StringList = new TStringList();
try {
 StringList->Add( “Animals” );
 StringList->Add( “Cats” );
 StringList->Add( “Flowers” );
 ComboBox1–>Width = 210;
 ComboBox1–>Items->Assign( StringList );
 ComboBox1–>ItemIndex = 0;
 Label1–>Caption = “Flowers has an index of “ +
   IntToStr( StringList->IndexOf( “Flowers” ) );
} __finally {
     StringList->Free();
}
```

To run the application

1. Save your project files; then choose **Run** ▶ **Run** to build and run the application. The form displays with the controls.
2. Click the Button. The strings 'Animals', 'Cars', and 'Flowers' display alphabetically in a list in the ListBox. The Label caption displays the message string: 'Flowers has an index value of 2.'
3. In the ComboBox, click the arrow to expand the drop-down list. The strings added in the TButton event handler appear.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

2.5.20 Creating a VCL Form Instance Using a Local Variable

A safe way to create a unique instance of a modal VCL form is to use a local variable in an event handler as a reference to a new instance. If you use a local variable, it doesn't matter whether the form is auto-created or not. The code in the event handler makes no reference to the global form variable. Using RAD Studio, the following procedure creates a modal form instance dynamically. It (optionally) removes the second form's invocation at startup.

Building this VCL application consists of the following steps:

1. Create the project directory.
2. Create two forms for the project.
3. Remove the second form's invocation at startup (optional).
4. Link the forms.
5. Create a control on the main form to create and display the modal form; then write the event handler.
6. Build and run the application.

To create the two forms

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** displays Form1.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** ▸ **Delphi Files** or C++Builder Projects and double-click the **Form** icon. The **VCL Forms Designer** displays Form2.

To optionally remove Form2's invocation at startup

1. Choose **Project** ▸ **Options** ▸ **Forms**. The **Project Options** dialog displays.
2. Select Form2 in the **Auto-create forms** list and click **[>]**. Form2 is moved to the **Available forms** list.
3. Click **OK** to close the dialog.

To link Form1 to Form2

1. Select Form1 and choose **File** ▸ **Use Unit** (Delphi) or Include Unit Hdr (C++). The **Uses Unit** dialog displays.
2. Select Form2 (the form that Form1 needs to reference) in the dialog.
3. Click **OK**.

For Delphi, a **uses** clause containing the unit name `Unit2` is placed in the implementation section of Unit1.

For C++, the `#include "Unit2.h"` directive is added to `Unit1.h`.

To display Form2 from Form1

1. Select Form1, if necessary; then, from the **Standard** page of the **Tool Palette**, place a TButton on the form.
2. In the **Object Inspector** with Button1 selected, double-click the OnClick event on the **Events** tab. The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. For Delphi, insert the cursor just above the event handler block and enter the following statement to define a local variable:

```
var FM: TForm2;
```

For C++, enter the following variable declaration:

```
TForm2 *FM;
```

4. Insert the cursor in the event handler block, and enter the following code:

```
FM := TForm2.Create(self);
FM.ShowModal;
FM.Free;
FM = new TForm2( this );
FM->ShowModal();
FM->Free();
```

To build and run the application

1. Save all files in the project; then choose **Run** ▸ **Run**. The application executes, displaying Form1.
2. Click the button. Form2 displays.

3. With Form2 displayed, attempt to click on Form1 to activate it. Nothing happens. Click the X in the upper right corner of Form2. Form2 closes and Form1 becomes the active form.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

Dynamically Creating a VCL Modeless Form (see page 121)

Displaying an Auto-Created VCL Form (see page 113)

2.5.21 Deleting Strings

Creating this VCL application consists of the following steps:

1. Create a VCL Form with Buttons and ListBox controls.
2. Write the code to add strings to a list.
3. Write the code to delete a string from the list.
4. Run the application.

To create a VCL Form with TButton and ListBox controls

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place two TButtons and a TListBox component on the form.
3. Select Button1 on the form.
4. In the **Object Inspector**, enter Add for the Name and Caption properties.
5. Select Button2 on the form.
6. In the **Object Inspector**, enter Delete for the Name and Caption properties.

To add strings to a list

1. Select the **Add** button on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.AddClick` (Delphi) or `TForm1::AddClick` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press return. This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declaration:

```
var
  MyList: TStringList;
```

For C++, enter the following variable declaration:

```
TStringList *MyList;
```

5. Insert the cursor within the code block, and type the following code:

```
MyList := TStringList.Create;
    try
      with MyList do
      begin
        Add('Mice');
        Add('Goats');
```

```
        Add('Elephants');
        Add('Birds');
        ListBox1.Items.AddStrings(MyList);
      end;
    finally
      MyList.Free;
    end;
MyList = new TStringList();
try {
 MyList->Add( “Mice” );
 MyList->Add( “Goats” );
 MyList->Add( “Elephants” );
 MyList->Add( “Birds” );
 ListBox1–>Items->AddStrings( MyList );
} __finally {
 MyList->Free();
}
```

To delete a string from the list

1. Select the **Delete** button on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.DeleteClick` (Delphi) or `TForm1::DeleteClick` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press `ENTER`. This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declaration:

```
var
  BIndex: Integer;
```

For C++, enter the following variable declaration:

```
int BIndex;
```

5. For Delphi, insert the cursor within the code block and type the following code:

```
with ListBox1.Items do
    begin
      BIndex := IndexOf('Elephants');
      Delete (BIndex);
    end;
BIndex = ListBox1–>Items->IndexOf( “Elephants” );
ListBox1–>Items->Delete( BIndex );
```

To run the application

1. Save your project files; then choose **Run** ▶ **Run** to build and run the application. The form displays with the controls.
2. Click the **Add** button. The strings 'Mice', 'Goats', 'Elephants', and 'Birds' display in the order listed.
3. Click the **Delete** button. The string 'Elephants' is deleted from the list.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

2.5.22 Displaying an Auto-Created VCL Form

Using RAD Studio, the following procedure creates a modal form at design time that is displayed later during program execution.

Building this VCL application consists of the following steps:

1. Create the project directory.
2. Create two forms for the project.
3. Link the forms.
4. Create a control on the main form to display the modal form; then write the event handler.
5. Build and run the application.

To create the two forms

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** displays Form1.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** ▸ **Delphi Files** or **File** ▸ **New** ▸ **Other** ▸ **C++Builder Files** and double-click the **Form** icon. The **VCL Forms Designer** displays Form2.

To link Form1 to Form2

1. For Delphi, select Form1 and choose **File** ▸ **Use Unit**. For C++, select Form1 and choose **File** ▸ **Include Unit Hdr**. The **Uses Unit** dialog displays.
2. Select Form2 (the form that Form1 needs to reference) in the dialog.
3. Click **OK**.

For Delphi, a **uses** clause containing the unit name Unit2 is placed in the implementation section of Unit1.

For C++, the `#include "Unit2.h"` directive is added to `Unit1.h`.

To display Form2 from Form1

1. Select Form1, if necessary; then, from the **Standard** page of the **Tool Palette**, place a button on the form.
2. In the **Object Inspector** with Button1 selected, double-click the OnClick event on the **Events** tab. The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. Enter the following event handling code:

```
Form2.ShowModal;
Form2->ShowModal();
```

To build and run the application

1. Save all files in the project; then choose **Run** ▸ **Run**. The application executes, displaying Form1.
2. Click the button. Form2 displays.
3. Click the **X** in the upper right corner of Form2. Form2 closes and Form1 becomes the active form.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

Dynamically Creating a VCL Modal Form (see page 120)

2.5.23 Displaying a Bitmap Image in a VCL Forms Application

This procedure loads a bitmap image from a file and displays it to a VCL form.

1. Create a VCL form with a button control.
2. Provide a bitmap image.
3. Code the button's onClick event handler to load and display a bitmap image.
4. Build and run the application.

To create a VCL form and button

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the **Tool Palette**, place a TButton component on the form.

To provide a bitmap image

1. Create a directory in which to store your project files.
2. Locate a bitmap image on your local drive, and copy it to your project directory.
3. Save all files in your project to your project directory.

To write the OnClick event handler

1. In the **Object Inspector**, double-click the Button1 OnClick event on the **Events** tab. The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
2. Enter the following event handling code, replacing `MyFile.bmp` with path to the bitmap image in your project directory:

```
Bitmap := TBitmap.Create;
try
  Bitmap.LoadFromFile('MyFile.bmp');
  Form1.Canvas.Brush.Bitmap := Bitmap;
  Form1.Canvas.FillRect(Rect(0,0,100,100));
finally
  Form1.Canvas.Brush.Bitmap := nil;
  Bitmap.Free;
end;
```

```
Graphics::TBitmap *Bitmap = new Graphics::TBitmap();
try {
  Bitmap->LoadFromFile( "..\\MyFile.bmp" );
  Form1->Canvas->Brush->Bitmap = Bitmap;
  Form1->Canvas->FillRect( Rect(0,0,100,100) );
} __finally {
  Form1->Canvas->Brush->Bitmap = NULL;
  Bitmap->Free();
}
```

Note: For C++ projects, the code assumes the target output directory is located in the project directory.

Tip: You can change the size of the rectangle to be displayed by adjusting the `Rect` parameter values.

3. For Delphi, add the following variable declaration in the **var** block:

```
Bitmap : TBitmap;
```

To run the program

1. Select **Run** ▶ **Run**.
2. Click the button to display the image bitmap in a 100 x 100-pixel rectangle in the upper left corner of the form.

See Also

VCL Overview (see page 36)

Drawing Straight Lines In a VCL Application (see page 119)

Drawing Rectangles and Ellipses in a VCL Application (see page 118)

Drawing a Polygon in a VCL Application (see page 117)

Placing A Bitmap Image In a Combo Box of a VCL Application (see page 135)

2.5.24 Displaying a Full View Bitmap Image in a VCL Forms Application

This procedure loads a bitmap image from a file and displays it in its entirety to a VCL form. The procedure uses the Height and Width properties of the Bitmap object to display a full view of the image.

1. Create a VCL form with a button control.
2. Provide a bitmap image.
3. Code the button's onClick event handler to load and display a bitmap image.
4. Build and run the application.

To create a VCL form and button

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the **Tool Palette**, place a button component on the form.
3. In the **Object Inspector**, enter Full View for the Caption property and FullView for the name property.

To provide a bitmap image

1. Create a directory in which to store your project files.
2. Locate a bitmap image on your local drive, and copy it to your project directory.
3. Save all files in your project to your project directory.

To write the OnClick event handler

1. In the **Object Inspector**, double-click the Button1 OnClick event on the **Events** tab. The **Code Editor** displays with the cursor in the `TForm1.FullViewClick` (Delphi) or `TForm1::FullViewClick` (C++) event handler block.
2. Enter the following event handling code, replacing `MyFile.bmp` with the name of the bitmap image in your project directory:

```
      Bitmap := TBitmap.Create;
      try
        Bitmap.LoadFromFile('MyFile.bmp');
        Form1.Canvas.Brush.Bitmap := Bitmap;
        Form1.Canvas.FillRect(Rect(0,0,Bitmap.Width,Bitmap.Height));
      finally
        Form1.Canvas.Brush.Bitmap := nil;
        Bitmap.Free;
      end;
```

```
Graphics::TBitmap Bitmap = new Graphics::TBitmap();
try {
  Bitmap->LoadFromFile( “..\\MyFile.bmp” );
  Form1–>Canvas->Brush->Bitmap = Bitmap;
  Form1–>Canvas->FillRect(
    Rect( 0, 0, Bitmap->Width, Bitmap->Height ) );
} __finally {
  Form1–>Canvas->Brush->Bitmap = NULL;
  Bitmap->Free();
}
```

Note: For C++ projects, the code assumes the target output directory is located in the project directory.

3. For Delphi, add the following variable declaration in the **var** block:

```
Bitmap : TBitmap;s
```

To run the program

1. Choose **Run** ▶ **Run**.
2. Click the button to display the image bitmap in a rectangle in the upper left corner of the form.

See Also

VCL Overview (see page 36)

Drawing Straight Lines In a VCL Forms Application (see page 119)

Displaying a Bitmap Image in a VCL Forms application (see page 114)

Drawing Rectangles and Ellipses in a VCL Forms Application (see page 118)

Drawing a Polygon in a VCL Forms Application (see page 117)

Placing A Bitmap Image In a Combo Box of a VCL Forms Application (see page 135)

2.5.25 Drawing a Polygon in a VCL Forms Application

This procedure draws a polygon in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw a polygon.
3. Build and run the application.

To create a VCL form

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. In the form view, click the form, if necessary, to display Form1 in the **Object Inspector**.

To write the OnPaint event handler

1. In the **Object Inspector**, click the **Events** tab.
2. Double-click the OnPaint event. The **Code Editor** displays with the cursor in the `TForm1.FormPaint` (Delphi) or `TForm1::FormPaint` (C++) event handler block.
3. Enter the following event handling code:

```
Canvas.Polygon ([Point(0,0), Point(0, ClientHeight),
Point(ClientWidth, ClientHeight)]);
TPoint points[] = { Point(0,0),
                    Point(0, ClientHeight),
                    Point(ClientWidth, ClientHeight) };
Canvas->Polygon( points, 3 );
```

To run the program

1. Select **Run** ▶ **Run**.
2. The applications executes, displaying a right triangle in the lower left half of the form.

See Also

VCL Overview (see page 36)

Drawing Straight Lines In a VCL Forms Application (see page 119)

Drawing Rectangles and Ellipses in a VCL Forms Application (see page 118)

Displaying a Bitmap Image in a VCL Forms Application (see page 114)

2.5.26 Drawing Rectangles and Ellipses in a VCL Forms Application

This procedure draws a rectangle and ellipse in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw a rectangle and ellipse.
3. Build and run the application.

To create a VCL form and place an image on it

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. In the form view, click the form, if necessary, to display Form1 in the **Object Inspector**.

To write the OnPaint event handler

1. In the **Object Inspector**, double-click the Form1 OnPaint event on the Events tab. The **Code Editor** displays with the cursor in the `TForm1.FormPaint` (Delphi) or `TForm1::FormPaint` (C++) event handler block.
2. Enter the following event handling code:

```
Canvas.Rectangle (0, 0, ClientWidth div 2, ClientHeight div 2);
Canvas.Ellipse (0, 0, ClientWidth div 2, ClientHeight div 2);
Canvas->Rectangle( 0, 0, ClientWidth / 2, ClientHeight / 2 );
Canvas->Ellipse( 0, 0, ClientWidth / 2, ClientHeight / 2 );
```

To run the program

1. Choose **Run** ▶ **Run**.
2. The applications executes, displaying a rectangle in the upper left quadrant, and an ellipse in the same area of the form.

See Also

VCL Overview (see page 36)

Drawing a Polygon in a VCL Forms Application (see page 117)

Displaying a Bitmap Image in a VCL Forms Application (see page 114)

2.5.27 Drawing a Rounded Rectangle in a VCL Forms Application

This procedure draws a rounded rectangle in a VCL form.

1. Create a VCL form and code the form's OnPaint event handler.
2. Build and run the application.

To create a VCL form

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. In the **Object Inspector**, click the **Events** tab.
3. Double-click the OnPaint event. The **Code Editor** displays with the cursor in the `TForm1.FormPaint` (Delphi) or `TForm1::FormPaint` (C++) event handler block.
4. Enter the following event handling code:

```
Canvas.RoundRect(0, 0, ClientWidth div 2,
  ClientHeight div 2, 10, 10);
Canvas->RoundRect( 0, 0, ClientWidth / 2, ClientHeight / 2, 10, 10 );
```

To run the program

1. Save all files in your project; then choose **Run** ▸ **Run**.
2. The application executes, displaying a rounded rectangle in the upper left quadrant of the form.

See Also

VCL Overview (see page 36)

Drawing Rectangles and Ellipses in a VCL Forms Application (see page 118)

2.5.28 Drawing Straight Lines In a VCL Forms Application

This procedure draws two diagonal straight lines on an image in a VCL form.

1. Create a VCL form.
2. Code the form's OnPaint event handler to draw the straight lines.
3. Build and run the application.

To create a VCL form and place an image on it

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. In the form view, click the form, if necessary, to display Form1 in the **Object Inspector**.

To write the OnPaint event handler

1. In the **Object Inspector**, double-click the Form1 OnPaint event on the **Events** tab. The **Code Editor** displays with the cursor in the `TForm1.FormPaint` event handler block.
2. Enter the following event handling code:

```
with Canvas do
begin
MoveTo(0,0);
LineTo(ClientWidth, ClientHeight);
MoveTo(0, ClientHeight);
LineTo(ClientWidth, 0);
end;
Canvas->MoveTo( 0, 0 );
Canvas->LineTo( ClientWidth, ClientHeight );
Canvas->MoveTo( 0, ClientHeight );
Canvas->LineTo( ClientWidth, 0 );
```

To run the program

1. Choose **Run** ▶ **Run**.
2. The applications executes, displaying a form with two diagonal crossing lines.

   Tip: To change the color of the pen to green, insert this statement following the first `MoveTo()` statement in the event handler code: `Pen.Color := clGreen;` (Delphi) `Canvas->Pen->Color = clGreen;` (C++). Experiment using other canvas and pen object properties. See "Using the properties of the Canvas object" in the Delphi 7 Developer's Guide.

See Also

VCL Overview (see page 36)

Building VCL Forms Applications With Graphics (see page 93)

Drawing Rectangles and Ellipses in a VCL Forms Application (see page 118)

Drawing a Polygon in a VCL Forms Application (see page 117)

Displaying a Bitmap Image in a VCL Forms Application (see page 114)

2.5.29 Dynamically Creating a VCL Modal Form

You may not want all your VCL application's forms in memory at once. To reduce the amount of memory required at load time, your application can create forms only when it needs to make them available for use. A dialog box, for example, needs to be in memory only during the time the user interacts with it. Using RAD Studio, the following procedure creates a modal form dynamically. The main difference between dynamically creating a form and displaying an auto-created VCL form is that you remove the second form's invocation at startup and write code to dynamically create the form.

Building this VCL application consists of the following steps:

1. Create the project directory.
2. Create two forms for the project.
3. Remove the second form's invocation at startup.
4. Link the forms.
5. Create a control on the main form to create and display the modal form; then write the event handler.
6. Build and run the application.

To create the two forms

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** displays Form1.
2. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** ▶ **Delphi Files** or **File** ▶ **New** ▶ **Other** ▶ **C++Builder Files** and double-click the **Form** icon. The **VCL Forms Designer** displays Form2.

To remove Form2's invocation at startup

1. Choose **Project** ▶ **Options** ▶ **Forms**. The **Project Options** dialog displays.
2. Select Form2 in the **Auto-create forms** list and click **[>]**. Form2 is moved to the **Available forms** list.
3. Click **OK** to close the dialog.

To link Form1 to Form2

1. For Delphi, select Form1 and choose **File** ▸ **Use Unit**. For C++, select Form1 and choose **File** ▸ **Include Unit Hdr** The **Uses Unit** dialog displays.
2. Select Form2 (the form that Form1 needs to reference) in the dialog.
3. Click **OK**. For Delphi, a **uses** clause containing the unit name Unit2 is placed in the implementation section of Unit1. For C++, the `#include "Unit2.h"` directive is added to `Unit1.h`.

To display Form2 from Form1

1. Select Form1, if necessary; then, from the **Standard** page of the **Tool Palette**, place a TButton on the form.
2. In the **Object Inspector** with Button1 selected, double-click the OnClick event on the **Events** tab. The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. Enter the following event handling code:

```
Form2 := TForm2.Create(self);
try
  Form2.ShowModal;
finally
  Form2.Free;
end;
Form2 = new TForm2( this );
try {
  Form2->ShowModal();
} __finally {
  Form2->Free();
}
```

To build and run the application

1. Save all files in the project; then choose **Run** ▸ **Run**. The application executes, displaying Form1.
2. Click the button. Form2 displays.
3. Click the **X** in the upper right corner of the form. Form2 closes and Form1 becomes the active form.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

Dynamically Creating a VCL Modeless Form (see page 121)

Displaying an Auto-Created VCL Form (see page 113)

2.5.30 Dynamically Creating a VCL Modeless Form

A modless form is a window that is displayed until it is either obscured by another window or until it is closed or minimuzed by the user. Using RAD Studio, the following procedure creates a modeless form dynamically.

Building this VCL application consists of the following steps:

1. Create the project directory.
2. Create two forms for the project.
3. Remove the second form's invocation at startup.
4. Link the forms.

5. Create a control on the main form to create and display the modal form; then write the event handler.
6. Build and run the application.

To create the two forms

1. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** displays Form1.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** ▸ **Delphi Files** or **File** ▸ **New** ▸ **Other** ▸ **C++Builder Files** and double-click the **Form** icon. The **VCL Forms Designer** displays Form2.

To remove Form2's invocation at startup

1. Choose **Project** ▸ **Options**. The **Project Options** dialog displays.
2. Select Form2 in the **Auto-create forms** list and click **[>]**. Form2 is moved to the **Available forms** list.
3. Click **OK** to close the dialog.

To link Form1 to Form2

1. For Delphi, select Form1 and choose **File** ▸ **Use Unit**. For C++, select Form1 and choose **File** ▸ **Include Unit Hdr**. The **Uses Unit** dialog displays.
2. Select Form2 (the form that Form1 needs to reference) in the dialog.
3. Click **OK**. For Delphi, a **uses** clause containing the unit name Unit2 is placed in the implementation section of Unit1. For C++, the `#include "Unit2.h"` directive is added to `Unit1.h`.

To display Form2 from Form1

1. Select Form1, if necessary; then, from the **Standard** page of the **Tool Palette**, place a button on the form.
2. In the **Object Inspector** with Button1 selected, double-click the OnClick event on the **Events** tab. The **Code Editor** displays with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++)event handler block.
3. Enter the following event handling code:

```
Form2 := TForm2.Create(self);
Form2.Show;
Form2 = new TForm2( this );
Form2->Show();
```

Note: If your application requires additional instances of the modeless form, declare a separate global variable for each instance. In most cases you use the global reference that was created when you made the form (the variable name that matches the Name property of the form).

To build and run the application

1. Save all files in the project; then choose **Run** ▸ **Run**. The application executes, displaying Form1.
2. Click the button. Form2 displays.
3. Click Form1. Form1 becomes the active form. Form2 displays until you minimize or close it.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

Displaying an Auto-Created VCL Form (see page 113)

Dynamically Creating a VCL Modal Form (see page 120)

2.5.31 Iterating Through Strings in a List

This VCL application first creates a list of strings. Then it iterates through the strings, changing all string characters to uppercase. It consists of the following steps:

1. Create a VCL Form with Buttons and TListBox controls.
2. Write the code to create a string list and add strings to it.
3. Write the code to iterate through the string list to process string characters.
4. Run the application.

To create a VCL Form with TButton and TListBox controls

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place two TButtons and a TListBox component on the form.
3. Select Button1 on the form.
4. In the **Object Inspector**, enter Add for the Name and Caption properties.
5. Select Button2 on the form.
6. In the **Object Inspector**, enter ToUpper for the Name and Caption properties.

To create a string list and add strings to it

1. Select the **Add** button on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.AddClick` (Delphi) or `TForm1::AddClick` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press ENTER. This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created, and type the following variable declaration:

```
var
  MyList: TStringList;
```

5. Insert the cursor within the code block, and type the following code:

```
MyList := TStringList.Create;
    try
      with MyList do
      begin
        Add('Mice');
        Add('Goats');
        Add('Elephants');
        Add('Birds');
        ListBox1.Items.AddStrings(MyList);
      end;
    finally
      MyList.Free;
    end;
TStringList *MyList = new TStringList();
try {
 MyList->Add(“Mice”);
 MyList->Add(“Goats”);
 MyList->Add(“Elephants”);
 MyList->Add(“Birds”);
 ListBox1–>Items->AddStrings( MyList );
```

```
} __finally {
 MyList->Free();
}
```

To change all characters to uppercase

1. Select the **ToUpper** button on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.ToUpperClick` (Delphi) or `TForm1::ToUpperClick` (C++) event handler block.
3. For Delphi, place the cursor before the `begin` reserved word; then press return. This creates a new line above the code block.
4. For Delphi, insert the cursor on the new line created and type the following variable declaration:

```
var
 Index: Integer;
```

5. Insert the cursor within the code block, and type the following code:

```
for Index := 0 to ListBox1.Items.Count - 1 do
    ListBox1.Items[Index] := UpperCase(ListBox1.Items[Index]);
for( int i = 0; i < ListBox1->Items->Count; i++ ) {
 ListBox1->Items[i] = UpperCase( ListBox1->Items[i] );
}
```

To run the application

1. Save your project files; then choose **Run** ▶ **Run** to build and run the application. The form displays with the controls.
2. Click the **Add** button. The strings 'Mice', 'Goats', 'Elephants', and 'Birds' display in the order listed.
3. Click the **ToUpper** button. The string characters display in uppercase.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

2.5.32 Building a Multithreaded Application

These are the essential steps to building a VCL Forms multithreaded application with a thread object using RAD Studio.

To drop a component on a form

1. Create a VCL form with a defined thread object.
2. Optionally initialize the thread.
3. Write the thread function.
4. Optionally write the cleanup code.

See Also

VCL Overview (see page 36)

Defining a Thread Object (see page 126)

Initializing a Thread (see page 130)

Writing the Thread Function (see page 134)

Writing Cleanup Code (see page 125)

2.5.33 Writing Cleanup Code

To clean up after your thread finishes executing

1. Centralize the cleanup code by placing it in the OnTerminate event handler. This ensures that the code gets executed.
2. Do not use any thread-local variables, because OnTerminate is not run as part of your thread.
3. You can safely access any objects from the OnTerminate handler.

See Also

VCL Overview (see page 36)

Building a Multithreaded Application (see page 124)

2.5.34 Avoiding Simultaneous Thread Access to the Same Memory

Use these basic techniques to prevent other threads from accessing the same memory as your thread:

1. Lock objects.
2. Use critical sections.
3. Use a multi-read exclusive-write synchronizer

To lock objects

1. For objects such as canvas that have a `Lock` method, call the `Lock` method, as necessary, to prevent other objects from accessing the object, and call `Unlock` when locking is no longer required.
2. Call `TThreadList.LockList` (Delphi) or `TThreadList::LockList()` (C++) to block threads from using the list object `TThreadList`, and call `TThreadList.UnlockList` when locking is no longer required.

   Note: You can safely make calls to `TCanvas.Lock` and `TThreadList.LockList`.

To use a critical section

1. Create a global instance of `TCriticalSection`.
2. Call the `Acquire` method to lock out other threads while accessing global memory.
3. Call the `Release` method so other threads can access the memory by calling `Acquire`. The following code has a global critical section variable `LockXY` that blocks access to the global variables `X` and `Y`. To use `X` or `Y`, a thread must surround that use with calls to the critical section such as shown here:

```
LockXY.Acquire;
try
      X := X + 1;
  Y := sin(X);
finally
  LockXY.Release
end;
LockXY->Acquire();
try {
  x++;
  y = sin( x );
```

```
} __finally {
  LockXY->Release();
}
```

Warning: Critical sections only work if every thread uses them to access global memory. Otherwise, problems of simultaneous access can occur.

To use the multi-read exclusive-write synchronizer

1. Create a global instance of `TMultiReadExclusiveWriteSynchronizer` that is associated with the global memory you want to protect.
2. Before any thread reads from the memory, it must call `BeginRead`.
3. At the completion of reading memory, the thread must call `EndRead`.
4. Before any thread writes to the memory, it must call `BeginWrite`.
5. At the completion of writing to the memory, the thread must call `EndWrite`.

   Warning: The multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Otherwise, problems of simultaneous access can occur.

See Also

VCL Overview (see page 36)

Writing the Thread Function (see page 134)

Using the Main VCL Thread (see page 131)

Waiting for Threads (see page 132)

Handling Exceptions (see page 129)

2.5.35 Defining the Thread Object

To define the thread object

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** ▶ **Delphi Files** or **File** ▶ **New** ▶ **Other** ▶ **C++Builder Files** and double-click the **Thread Object** icon. The **New Thread Object** dialog displays.
2. Enter a class name, for example, `TMyThread`.
3. Optionally check the **Named Thread** check box, and enter a name for the thread, for example, MyThreadName.

   Tip: Entering a name for Named Thread makes it easier to track the thread while debugging.

4. Click **OK**.

The **Code Editor** displays the skeleton code for the thread object.

The code generated for the new unit will look like this if you named your thread class TMyThread.

```
unit Unit1;

interface

uses
  Classes;

type
  TMyThread = class(TThread)
  private
```

```
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation

{ Important: Methods and properties of objects in visual components can only be
  used in a method called using Synchronize, for example,

      Synchronize(UpdateCaption);

  and UpdateCaption could look like,

    procedure TMyThread.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }

{ TMyThread }

procedure TMyThread.Execute;
begin
  { Place thread code here }
end;

end.
```

Adding a name for the thread generates additional code for the unit. It includes the Windows unit, adds the procedure (Delphi) or function (C++) `SetName`, and adds the record `TThreadNameInfo` (Delphi) or struct `THREADNAME_INFO` (C++). The name is assigned to the `FName` field, as shown here:

```
unit Unit1;

interface

uses
  Classes {$IFDEF MSWINDOWS} , Windows {$ENDIF};

type
  TMyThread = class(TThread)
  private
    procedure SetName;
  protected
    procedure Execute; override;
  end;

implementation

{ Important: Methods and properties of objects in visual components can only be
  used in a method called using Synchronize, for example,

      Synchronize(UpdateCaption);

  and UpdateCaption could look like,

    procedure TMyThread.UpdateCaption;
    begin
      Form1.Caption := 'Updated in a thread';
    end; }

{$IFDEF MSWINDOWS}
type
  TThreadNameInfo = record
    FType: LongWord;     // must be 0x1000
    FName: PChar;        // pointer to name (in user address space)
```

```
    FThreadID: LongWord; // thread ID (-1 indicates caller thread)
    FFlags: LongWord;    // reserved for future use, must be zero
  end;
{$ENDIF}

{ TMyThread }

procedure TMyThread.SetName;
{$IFDEF MSWINDOWS}
var
  ThreadNameInfo: TThreadNameInfo;
{$ENDIF}
begin
{$IFDEF MSWINDOWS}
  ThreadNameInfo.FType := $1000;
  ThreadNameInfo.FName := 'MyThreadName';
  ThreadNameInfo.FThreadID := $FFFFFFFF;
  ThreadNameInfo.FFlags := 0;

  try
    RaiseException( $406D1388, 0, sizeof(ThreadNameInfo) div sizeof(LongWord), @ThreadNameInfo
);
  except
  end;
{$ENDIF}
end;

procedure TMyThread.Execute;
begin
  SetName;
  { Place thread code here }
end;

end.
// Unit1.h

#ifndef Unit1H
#define Unit1H
#include <Classes.hpp>
class TMyThread : public TThread
{
  typedef struct tagTHREADNAME_INFO
  {
    DWORD dwType;
    LPCSTR szName;
    DWORD dwThreadID;
    DWORD dwFlags;
  } THREADNAME_INFO;
private:
  void SetName();
protected:
  void __fastcall Execute();
public:
  __fastcall TMyThread(bool CreateSuspended);
};

#endif
// Unit1.cpp
#include "Unit3.h"
#pragma package(smart_init)

__fastcall TMyThread::TMyThread(bool CreateSuspended)
  : TThread(CreateSuspended)
{
}
```

```
void TMyThread::SetName()
{
  THREADNAME_INFO info;
  info.dwType = 0x1000;
  info.szName = "TMyThreadName";
  info.dwThreadID = -1;
  info.dwFlags = 0;

  __try
  {
     RaiseException( 0x406D1388, 0, sizeof(info)/sizeof(DWORD),(DWORD*)&info; );
  }
  __except (EXCEPTION_CONTINUE_EXECUTION)
  {
  }
}

void __fastcall TMyThread::Execute()
{
  SetName();
  //---- Place thread code here ----
}
```

See Also

VCL Overview (see page 36)

Initializing the Thread (see page 130)

Writing the Thread Function (see page 134)

Writing Cleanup Code (see page 125)

2.5.36 Handling Exceptions

To handle exceptions in the thread function

1. Add a **try...except** block to the implementation of your `Execute` method.
2. Code the logic such as shown here:

```
procedure TMyThreadExecute;
begin
  try
    while not Terminated do
      PerformSomeTask;
  except
    {do something with exceptions}
  end;
end;
void __fastcall TMyThread::Execute() {
  try {
    while( !Terminated() ) {
      // perform tasks
    }
  } catch(...) { // catch specific exceptions first
    // exception–handling code
  }
}
```

See Also

VCL Overview (see page 36)

Writing the Thread Function (see page 134)

Using the Main VCL Thread (see page 131)

Avoiding Simultaneous Thread Access to the Same Memory (see page 125)

Waiting for Threads (see page 132)

2.5.37 Initializing a Thread

To initialize a thread object

1. Assign a default thread priority.
2. Indicate when the thread is freed.

To assign a default priority

1. Assign a default priority to the thread from the values listed in the table below. Use a high-priority to handle time critical tasks, and a low priority to perform other tasks.

Value	Priority
tpIdle	The thread executes only when the system is idle. Windows won't interrupt the other threads to execute a thread with tpIdle priority.
tpLowest	The thread's priority is two points below normal.
tpLower	The thread's priority is one point below normal.
tpNormal	The thread has normal priority.
tpHigher	The thread's priority is one point above normal.
tpHighest	The thread's priority is two points above normal.
tpTimeCritical	The thread gets highest priority.

2. Override the `Create` method (Delphi) or default constructor (C++) of the thread class by adding a new constructor to the declaration.
3. Code the constructor. The following is an example for a low-priority thread:

```
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  inherited Create(CreateSuspended);
  Priority := tpIdle;
end;
TMyThread::TMyThread( bool CreateSuspended ) : TThread( CreateSuspended ) {
  Priority = tpIdle;
}
```

4. Indicate whether the thread should be freed automatically when it finishes executing.

   Warning: Boosting the thread priority of a CPU intensive operation may starve other threads in the application. Only apply priority boosts to threads that spend most of their time waiting for external events.

To indicate when a thread is freed

1. Set the `FreeOnTerminate` property to **true**, unless the thread must be coordinated with other threads.
2. If the thread requires coordination with another thread, set `FreeOnTerminate` to **false**; then explicitly free the first thread from the second.

See Also

VCL Overview (see page 36)

Defining a Thread Object (see page 126)

Writing the Thread Function (see page 134)

Writing Cleanup Code (see page 125)

2.5.38 Using the Main VCL Thread

Using the main VCL thread consists of the following basic steps:

1. Create a separate routine to handle Windows messages received by components in your application.
2. Call CheckSynchronize periodically.
3. Declare thread-local variables, as necessary, for exclusive use by your thread.

To create a separate routine

1. Write a main thread routine that handles accessing object properties and executing object methods for all objects in your application.
2. Call the routine using the `TThread.Synchronize` (Delphi) or `TThread::Synchronize` method. The following code is an example of how to call a method using `Synchronize`

```
procedure TMyThread.PushTheButton
begin
  Button1.Click;
end;
procedure TMyThread.Execute;
begin
  ...
  Synchronize(PushThebutton);
  ...
end;
void TMyThread::PushTheButton() { Form1->Button1->Click(); }
void __fastcall TMyThread::Execute() {
  ...
  Synchronize( (TThreadMethod)&PushTheButton );
  ...
}
```

`Synchronize` waits for the main thread to enter the message loop and then executes the passed method.

Note: Because `Synchronize` uses a message loop, it does not work in console applications. For console applications, use other mechanisms, such as critical sections, to protect access to VCL objects.

To call CheckSynchronize

1. Call `CheckSynchronize` periodically within the main thread to enable background threads to synchronize execution with the main thread.
2. To ensure the safety of making background thread calls, call `CheckSynchronize` when the application is idle, for example, from an `OnIdle` event handler.

To use a thread-local variable

1. Identify variables that you want to make global to all the routines running in your thread but not shared by other instances of the same thread class.

2. For Delphi, declare these variables in a **threadvar** section, for example,

```
threadvar
  x: integer;
```

For C++, declare these variables with the __**thread** modifier:

```
int __thread x;
```

Note: Use the threadvar

section for global variables only. Do not use it for Pointer and Function variables or types that use copy-on-write semantics, such as long strings.

Note: For C++, if you initialize a __thread

variable, you must initialize it to a constant expression. For example, `int __thread foo = 3;` is a legal statement, but `int __thread foo = get_value();` is not permitted because the initialization occurs at runtime.

See Also

VCL Overview (see page 36)

Writing the Thread Function (see page 134)

Avoiding Simultaneous Thread Access to the Same Memory (see page 125)

Waiting for Threads (see page 132)

Handling Exceptions (see page 129)

2.5.39 Waiting for Threads

The following are procedures that can be used to wait for threads.

- Wait for a thread to finish executing.
- Wait for a task to complete.
- Check if another thread is waiting for your thread to terminate.

To wait for a thread to finish executing

1. Use the `WaitFor` method of the other thread.
2. Code your logic. For example, the following code waits for another thread to fill a thread list object before accessing the objects in the list:

```
if ListFillingThread.WaitFor then
begin
  with ThreadList1.LockList do
  begin
               for I := 0 to Count - 1 do
      ProcessItem(Items[I];
  end;
  ThreadList1.UnlockList;
end;
if( ListFillingThread->WaitFor() ) {
  TList* list = ThreadList1->LockList();
  for( int i = 0; i < list->Count; i++ ) {
    DoSomething( list->Items[i] );
  }
  ThreadList1->UnlockList();
}
```

To wait for a task to complete

1. Create a TEvent object of global scope.
2. When a thread completes an operation other threads are waiting for, have the thread call `TEvent.SetEvent`.
3. To turn off the signal, call `TEvent.ResetEvent`.

The following example is an OnTerminate event handler that uses a global counter in a critical section to keep track of the number of terminating threads. When `Counter` reaches 0, the handler calls the `SetEvent` method to signal that all processes have terminated:

```
procedure TDataModule.TaskTerminateThread(Sender: TObject);
begin
  ...
  CounterGuard.Acquire; {obtain a lock on the counter}
  Dec(Counter); {decrement the global counter variable}
  if Counter = 0 then
    Event1.SetEvent; {signal if this is the last thread}
  Counter.Release; {release the lock on the counter}
  ...
end;
void __fastcall TDataModule::TaskThreadTerminate( TObject *Sender ) {
  ...
  CounterGuard->Acquire(); // lock the counter
  if( --Counter == 0 )      // decrement counter
    Event1->SetEvent();    // signal if this is the last thread
  CounterGuard->Release(); // release lock
}
```

The main thread initializes `Counter`, launches the task threads, and waits for the signal that they are all done by calling the `TEvent::WaitFor` method. `WaitFor` waits a specified time period for the signal to be set and returns one of the values in the table below.

The following code example shows how the main thread launches the task threads and resumes when they have completed.

```
Event1.ResetEvent; {clear the event before launching the threads}
for i := 1 to Counter do
  TaskThread.Create(False); {create and launch the task threads}
if Event1.WaitFor(20000) <> wrSignaled then
  raise Exception;
{continue with main thread}
Event1->ResetEvent();  // clear the event before launching threads
for( int i = 0; i < Counter; i++ ) {
  new TaskThread( false );
if( Event1->WaitFor(20000) != wrSignaled )
  throw Exception;
// now continue with the main thread
```

Note: If you do not want to stop waiting for an event handler after a specified time period, pass the `WaitFor` method a parameter value of INFINITE. Be careful when using INFINITE, because your thread will hang if the anticipated signal is never received.

To check if another thread is waiting on your thread to terminate

1. In your `Execute` procedure, implement the `Terminate` method by checking and responding to the `Terminated` property.
2. This is one way to code the logic:

```
procedure TMyThread.Execute;
begin
  while not Terminated do
    PerformSomeTask;
end;
void __fastcall TMyThread::Execute() {
  while( !Terminated )
```

```
    DoSomething();
}
```

WaitFor return values

Value	Meaning
wrSignaled	The signal of the event was set.
wrTimeout	The specified time elapsed without the signal being set.
wrAbandoned	The event object was destroyed before the timeout period elapsed.
wrError	An error occurred while waiting.

See Also

VCL Overview (see page 36)

Writing the Thread Function (see page 134)

Using the Main VCL Thread (see page 131)

Avoiding Simultaneous Thread Access to the Same Memory (see page 125)

Handling Exceptions (see page 129)

2.5.40 Writing the Thread Function (Procedure)

The Execute method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program, because you must make sure that you do not overwrite memory that is used by other processes in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

To implement Execute, coordinate thread execution by

1. Synchronizing with a main VCL thread.
2. Avoiding simultaneous access to the same memory.
3. Waiting for threads.
4. Handling exceptions.

See Also

VCL Overview (see page 36)

Using the Main VCL Thread (see page 131)

Avoiding Simultaneous Thread Access to the Same Memory (see page 125)

Waiting for Threads (see page 132)

Handling Exceptions (see page 129)

2.5.41 Placing A Bitmap Image in a Control in a VCL Forms Application

This procedure adds a bitmap image to a combo box in a VCL forms application.

1. Create a VCL form.
2. Place components on the form.
3. Set component properties in the Object Inspector.
4. Write event handlers for the component's drawing action.
5. Build and run the application.

To create a VCL form with a TComboBox component

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Win32** page of the **Tool Palette**, place an TImageList component on the form.
3. From the **Standard** page of the **Tool Palette**, place a TComboBox on the form.

To set the component properties

1. Select ComboBox1 in the form.
2. In the **Object Inspector**, set the Style property drop-down to csOwnerDrawFixed.
3. In the **Object Inspector**, click the ellipsis next to the Items property. The **String List Editor** displays.
4. Enter a string you would like to associate with the bitmap image, for example, MyImage; then click **OK**.
5. Double-click ImageList1 in the form. The ImageList editor displays.
6. Click the **Add** button to display the **Add Images** dialog.
7. Locate a bitmap image to display in the Combo box. To locate an image, you can search for *.bmp images on your local drive. Select a very small image such as an icon. Copy it to your project directory, and click **Open**. The image displays in the ImageList editor.
8. Click **OK** to close the editor.

To add the event handler code

1. In the VCL form view, select ComboBox1.
2. In the Object Inspector, click the Events page, and double-click the OnDrawItem event. The **Code Editor** displays with cursor in the code block of the `ComboBox1DrawItem` (Delphi) or `ComboBox1::DrawItem` (C++) event handler.
3. Enter the following code for the event handler:

```
Combobox1.Canvas.FillRect(Rect);
ImageList1.Draw(ComboBox1.Canvas, Rect.Left, Rect.Top, Index);
Combobox1.Canvas.TextOut(Rect.Left+ImageList1.Width+2,
  Rect.Top, ComboBox1.Items[Index]);
```

```
ComboBox1->Canvas->FillRect( Rect );
ImageList1->Draw( ComboBox1->Canvas, Rect.Left, Rect.Top, Index );
ComboBox1->Canvas->TextOut( Rect.Left + ImageList1->Width + 2,
                            Rect.Top,
                            ComboBox1->Items[Index] );
```

To run the program

1. Choose **Run** ▸ **Run**. The applications executes, displaying a form with a combo box.
2. Click the combo box drop-down. The bitmap image and the text string display as a list item.

See Also

VCL Overview (see page 36)

Building VCL Forms Applications With Graphics (see page 93)

2.5.42 Reading a String and Writing It To a File

Creating this VCL application consists of the following steps:

1. Create a VCL Form with a button control.
2. Write the code to read the string and write it to a file.
3. Run the application.

To create a VCL Form

1. Create a directory in which to store your project files.
2. Choose **File** ▸ **New** ▸ **Other** ▸ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
3. From the **Standard** page of the Tool palette, place a TButton component on the form.

To read and write a string

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. For Delphi. place the cursor before the `begin` reserved word; then press return. This creates a new line above the code block.
4. Type the following variable declarations:

```
TFileStream *fs const AnsiString str = "Hello";
```

5. Insert the cursor within the code block, and type the following code:

```
fs = new TFileStream("temp.txt", fmCreate);
 fs->Write ((void*)str.c_str(), str.fmCreate);
```

To run the "Hello world" application

1. Save your project files; then choose **Run** ▸ **Run** to build and run the application. The form displays with a button called **Button1**.
2. Click **Button1**.
3. Use a text editor to open the newly created file temp.txt, which is located in your project directory. The string 'Hello' displays in the file.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

2.5.43 Renaming Files

Creating this VCL application consists of the following steps:

1. Create a project directory containing a file to rename.
2. Create a VCL Form with button and label controls.
3. Write the code to rename the file.
4. Run the application.

To set up your project directory and a text file to copy

1. Create a directory in which to store your project files.
2. Either create or copy a text file to your project directory; then save it as MyFile.txt.

To create a VCL Form with a button and label

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Standard** page of the Tool palette, place a TButton component on the form.
3. From the **Standard** page of the Tool palette, place a TLabel component on the form.

To write the rename file procedure

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the `TForm1.Button1Click` (Delphi) or `TForm1::Button1Click` (C++) event handler block.
3. At the cursor, type the following code:

```
if not RenameFile('MyFile.txt', 'YourFile.txt') then
Label1.Caption := 'Error renaming file!';
if( !RenameFile( "..\\MyFile.txt", "..\\YourFile.txt" )
  Label1->Caption = “Error renaming file”;
// the file parameters assume the target output directory is in your project directory
```

Note: You cannot rename (move) a file across drives using RenameFile. You would need to first copy the file and then delete the old one. In the runtime library, RenameFile is a wrapper around the Windows API MoveFile function, so MoveFile will not work across drives either.

To run the application

1. Save your project file; then choose **Run** ▶ **Run** to build and run the application. The form displays.
2. Click the button; If no message displays in the Label, check the file name in your project directory. MyFile.txt should is renamed as YourFile.txt.
3. If the caption label displays the error message, recheck your event handler code.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

2.5.44 Adding and Sorting Strings

Creating this VCL application consists of the following steps:

1. Create a VCL Form with Button, Label, and TListBox controls.
2. Write the code to add and sort strings.
3. Run the application.

To create a VCL Form with Button, Label, and ListBox controls

1. Choose File ▸ New ▸ Other ▸ Delphi Projects or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** displays.
2. From the **Standard** category of the **Tool Palette**, place a TButton, TLabel, and TListBox component on the form.

To write the copy stream procedure

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the TForm1.Button1Click event handler block.
3. For Delphi, place the cursor before the `begin` reserved word and press ENTER. This creates a new line above the code block.
4. Type the following variable declarations:

```
var
  MyList: TStringList;
  Index: Integer;
TStringList *MyList;
int Index;
```

5. Insert the cursor within the code block, and type the following code:

```
MyList := TStringList.Create;
try
  MyList.Add('Animals');
  MyList.Add('Flowers');

  MyList.Add('Cars');

  MyList.Sort;
    if MyList.Find('Flowers', Index) then
    begin
      ListBox1.Items.AddStrings(MyList);
      Label1.Caption := 'Flowers has an index value of ' + IntToStr(Index);
    end;
finally
    MyList.Free;
end;
MyList = new TStringList();
try {
 MyList->Add( "Animals" );
 MyList->Add( "Flowers" );
 MyList->Add( "Cars" );
 MyList->Sort();
 if( MyList->Find( "Flowers", Index ) {
   ListBox1->Items->AddStrings( MyList );
   Label1->Caption = "Flowers has an index of " +
                     IntToStr( Index );
 }
} __finally {
```

```
 MyList->Free();
}
```

Note: Find will only work on sorted lists. Use IndexOf on unsorted lists.

To run the application

1. Save your project files; then choose **Run** ▸ **Run** to build and run the application. The form displays with the controls.
2. Click the Button. The strings 'Animals', 'Cars', and 'Flowers' display alphabetically in a list in the ListBox. The Label caption displays the message string: 'Flowers has an index value of 2.'

See Also

VCL Overview (see page 36)

2.5.45 Creating a VCL Forms ActiveX Button

Like a Delphi control, an ActiveX control generates program code when you place the component on a form or other logical container in the IDE. The main difference between an ActiveX control and a Delphi control is that an ActiveX control is language independent. You can create ActiveX controls for deployment to a variety of programming environments on Windows, not just Delphi or C++Builder.

This procedure uses the VCL forms ActiveX wizard to create an ActiveX control. To test the control, you can install it on your machine as a VCL component in the IDE. To install the control, you first need to create a package for it. This procedure consists of the following major steps:

Metaprocedure for Creating a VCL Forms ActiveX Button

1. Create an ActiveX library project for an ActiveX button control.
2. Register the ActiveX button so its icon can be displayed in the toolbar.
3. Create a package for the ActiveX button.
4. Install the package.
5. Test the ActiveX button.

To create an ActiveX library project for an ActiveX button control

1. Create a directory on your local drive for the ActiveX project. Give it a name that you can find easily, for example, ActiveX.
2. Choose **File** ▸ **New** ▸ **Other** and select the **ActiveX** page in the **New Items** dialog box.
3. In the **ActiveX** folder, double-click **ActiveX Library**. This creates a Dynamic Link Library [DLL] project that you can use to host in-process ActiveX Objects.
4. Choose **File** ▸ **New** ▸ **Other** again.
5. On the **ActiveX** page, double-click **ActiveX Control**. The **ActiveX Control Wizard** displays.
6. In the **Component Name** drop-down, select **TButton**.
7. By default, **ButtonX** displays as the **New ActiveX Name**. Rename ButtonX to the name you want displayed for your ActiveX button, for example, MyActiveXButton.

   Note: Modifications you make to the name update the Implementation Unit and Project Name. Leave the remaining fields with default values.

8. Click **OK**. The wizard generates the code needed to implement the ActiveX control and adds the code to the project. If the project is already an ActiveX library, the wizard adds the control to the current project.

Note: If the project is not already an ActiveX library, a Warning

dialog displays and asks you if you want to start a new ActiveX library project.

9. Click **OK** to start the new ActiveX Library project.

To register the ActiveX button

1. Build the project and save all files to your ActiveX project directory. Dismiss the warning about debugging. The project builds and creates an OCX file in your project directory.
2. Choose **Run** ▶ **Register ActiveX Server** to register the ActiveX button. A dialog box displays a message indicating that registration was successful and it shows the path to the resulting OCX file.
3. Click **OK**.

To create a new package for the ActiveX button

1. Choose **File** ▶ **New** ▶ **Other** to create a new package. The **New Items** dialog displays.
2. Double-click Package on the **New** page to display the **Package - package.dpk** dialog and click **Add**.
3. On the **Add unit** tab of the **Add** dialog, browse to your project directory.
4. Select the `ButtonXControl1_TLB.pas` file, and click **Open**.
5. Click **OK** to add the file to the package and return to the **Package - package.dpk** dialog. The **Package - package.dpk** dialog displays showing the files in the package and two required files: rtl.dcp and vcl.dcp.

To add the required files and install the package

1. In the **Package - package.dpk** dialog, select rtl.dcp, and click **Add**.
2. On the **Add unit** tab of the **Add** dialog, browse to the Lib directory in Delphi, select the rtl.dcp file, and click **Open**; then click **OK** on the **Add** dialog.
3. In the **Package - package.dpk** dialog, select vcl.dcp, and click **Add**.
4. On the **Add unit** tab of the **Add** dialog, browse to the Lib directory in Delphi, select the vcl.dcp file, and click **Open**; then click **OK** on the **Add** dialog.
5. In the **Package - package.dpk** dialog, click **Compile** to compile the package. A dialog displays, indicating that the package has been installed. Click **OK**.
6. Click the X in the upper right corner of the **Package - package.dpk** dialog to close it. You are prompted to save the package.
7. Save the package to your projects directory.

To test the button

1. Choose **File** ▶ **New Application**.
2. From the **ActiveX** page of the **Tool Palette**, locate your button and place it on the form. The button displays on the form.

See Also

VCL Overview (see page 36)

Creating a VCL Forms ActiveX Active Form (see page 140)

2.5.46 Creating a VCL Forms ActiveX Active Form

Like a Delphi control, an ActiveX control generates program code when you place the component on a form or other logical container in the IDE. The main difference between an ActiveX control and a Delphi control is that an ActiveX control is language independent. You can create ActiveX controls for deployment to a variety of programming environments on Windows, not just

Delphi or C++Builder, for example.

This procedure uses the VCL forms ActiveX Active Form wizard to create an Active Form containing two components. To test the control, you can deploy it to the Web. This procedure consists of the following major steps:

1. Create an ActiveX library project for an ActiveX Active Form.
2. Add controls to the Active Form.
3. Add event handling code for the controls.
4. Deploy the project to the Web.
5. Display the form and test the controls in your Web browser.

To create an Active X library project for an ActiveX Active Form

1. Create a directory on your local drive for the ActiveX project. Give it an easy to find name, for example, ActiveX.
2. Create a second directory to contain the ActiveX component and an HTML file for deploying the Active Form to your Microsoft Internet Explorer Web browser. Name this directory ActiveX_Deploy.
3. Choose **File** ▸ **New** ▸ **Other** and select the **ActiveX** page in the **New Items** dialog.
4. On the **ActiveX** page, double-click Active Form. The **Active Form Wizard** displays.
5. Accept the default settings and click **OK**. The wizard generates the code needed to implement the ActiveX control and adds the code to the project. If the project is already an ActiveX library, the wizard adds the control to the current project.

   Note: If the project is not already an ActiveX library, a Warning

   dialog displays and asks you if you want to start a new ActiveX library project.
6. Click **OK** to start a new ActiveX library project. An ActiveX Active Form displays.

To add some functionality to the Active Form

1. From the **Standard** page of the **Tool Palette**, add TEdit and TButton components to the form.
2. Select the button.
3. On the **Events** tab in the **Object Inspector**, double-click the OnClick event. The **Code Editor** opens with the cursor in place in the `TActiveFormX.Button1Click` (Delphi) or `TActiveFormX::Button1Click()` (C++) event handler block. Enter the following code at the cursor:

```
ShowMessage(Edit1.text);
ShowMessage( Edit1->Text )
```

4. Save the project files to your ActiveX directory.

   Note: To deploy the Active form to your Web, use the Deployment Manager

   . Click **File** ▸ **New** ▸ **Other** ▸ **Deployment**.

To test the Active Form

1. Launch your browser.
2. Choose **File** ▸ **Open**, and browse to the ActiveX_Deploy directory.
3. Double-click the HTML file to launch it in the browser window. The Active Form displays in the browser window.
4. Click the button. A pop-up dialog displays the text in the Edit box.
5. Change the text, and click the button again. The new text you entered displays in the pop-up.

See Also

VCL Overview (see page 36)

Creating a VCL Forms ActiveX Button (see page 139)

2.5.47 Building a VCL Forms Web Browser Application

Creating the Web browser application consists of the following steps:

1. Create a VCL Form with a button control.
2. Add a TWebBrowser component to the form.
3. Add controls to enter a URL and launch the browser.
4. Write the code to launch the browser when a button is clicked.
5. Run the application.

To create a VCL Form

1. Choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** or C++Builder Projects and double-click the **VCL Forms Application** icon. The **VCL Forms Designer** is displayed.
2. From the **Internet** page of the **Tool Palette**, place a TWebBrowser component on the form.
3. With the TWebBrowser component selected on the form, drag the handles to adjust the size of the browser window. Leave some space on the form above the TWebBrowser to add a URL entry window. If the window is not large enough to display a browser page in its entirety, the TWebBrowser component adds scrollbars when you run the application and launch the browser window.
4. From the **Standard** page of the **Tool Palette**, place a TMemo component on the form. With the TMemo component selected on the form, drag the handles to adjust the size to accommodate a user-entered URL.
5. From the **Standard** page of the **Tool Palette**, place a Label component on the form.
6. Select the Label, and in the **Object Inspector**, enter URL: as the Label caption.
7. From the **Standard** page of the **Tool Palette**, place a TButton component on the form.
8. Select the Button, and in the **Object Inspector**, enter OK as the TButton caption.

To code a button click event that launches the browser

1. Select Button1 on the form.
2. In the **Object Inspector**, double-click the OnClick action on the **Events** tab. The **Code Editor** displays, with the cursor in the Button1Click event handler block.
3. Type the following code:

```
WebBrowser1.Navigate(WideString(Memo1.Text));
WebBrowser1->Navigate( WideString( Memo1->Text ) );
```

To run the application

1. Choose **Run** ▶ **Run** to build and run the application.
2. Enter a URL to a Web page in the memo window; then click the button. The browser launches in the TWebBrowser window.

See Also

VCL Overview (see page 36)

Building a VCL Forms Application (see page 99)

2.5.48 Creating an Application that Uses Ribbon Controls

This procedure describes how to create an application that uses ribbon controls. The core ribbon functionality is derived from the TRibbon component. While the ribbon uses other components, none of the core components are registered on the tool palette.

Components:

1. **TRibbon:** Main visual component that provides most functionality.
2. **TRibbonApplicationMenuBar:** Component that provides the functionality of the application menu.
3. **TRibbonQuickAccessToolbar:** Component that provides the functionality of the Quick Access Toolbar
4. **TRibbonPage:** Component that represents the page of the ribbon that is currently visible
5. **TRibbonGroup:** Component that all of the pages commands are displayed in. Commands must be placed in a group.

To create a new application that uses ribbon controls

1. Create a new Delphi Win32 VCL application (select **New ▸ VCL Forms Application — Delphi for Win32**).
2. Drop a TActionManager component onto the form.
3. Drop a TRibbon component onto the form.
4. Right-click the TRibbon component and select the **Add Application Menu** item.
5. Right-click the TRibbon component again and select the **Add Quick Access Toolbar** item.

   Note: During initial configuring of the ribbon you might see a dialog saying that the current style selected in the TActionManager.Style is not a valid ribbon style. Change the TRibbonManager.Style property to be the **Ribbon – Luna**. Applicable styles for use with the ribbon have the Ribbon prefix. By default three ribbon styles are implemented. These ribbon styles also work with the default ActionManager toolbar and menu bar controls. The available ribbon Styles are:

- Luna
- Obsidian
- Silver

See Also

VCL Overview (see page 36)

Adding Commands to the Ribbon (see page 144)

TRibbon

TRibbon

TRibbonApplicationMenuBar

Caption

TRibbonQuickAccessToolbar

TRibbonPage

TRibbonGroup

2.5.49 Adding Commands to the Ribbon

This topic follows in sequence the creation of a ribbon application using either the Ribbon Application Wizard or the manual steps described in Creating an Application that Uses Ribbon Controls (see page 143).

This topic assumes that you are familiar with the TActionManager component and the components associated with its use. Numerous new properties have been added to help support the ribbon requirements. Many existing properties have no effect when modified on a command that is displayed on the ribbon.

For instance:

- Small buttons always display their glyph to the left of the caption.
- Large buttons always display their glyph at the top.
- A Large button must have a caption. The caption can be a maximum of two lines, and the button width is resized accordingly. A large button's height is fixed to cover the height of the group where the command is displayed.
- Only small buttons with no caption can be displayed in groups.

New Properties

CommandStyle: The CommandStyle determines the type of control that is used to represent this command on the ribbon. The default style is **csButton**. CommandStyle is used extensively for determining the control type that is used for the command except under one exception.

```
type TCommandStyle = (csButton, csMenu, csSeparator, csText, csGallery, csComboBox, csControl, csCustom);
```

Style of the Command on the Ribbon Group

csButton	Command is a button.
csComboBox	Command displays an Office 2007 style combobox. See following Note.
csControl	Command has a TControl descendant associated with it. An example is dropping a TEdit component onto a ribbon Group.
csCustom	User-defined.
csGallery	Command is a gallery or when selected, it displays a gallery.
csMenu	Command is a menu item
csSeparator	Command is a separator with a text description.
csText	Command only displays text and has no associated control.

Note: You can only use the csComboBox CommandStyle if you first place a TRibbonComboBox from the tool palette onto your ribbon Group.

CommandProperties: CommandProperties is a dynamic property class. The published properties of this property (it is a TPersistent descendant) vary depending on the CommandStyle that has been selected. If you select the **csButton** CommandStyle, the CommandProperties class used is called TButtonProperties. This property class publishes properties that are specific to the csButton CommandStyle.

An example is ButtonSize, which allows you to indicate if the button is a large button or a small button. There is no need for a csMenu command to have a ButtonSize property because the menu does not react to a size variance.

All CommandProperties classes descend from **TCommandProperties**. If you are using the csCustom CommandStyle, you must descend from the TCommandProperties class.

Properties that can be available in the CommandProperties property depending on your selected CommandStyle

csButton	ButtonSize
	Size of the button:
	TButtonSize = (bsSmall, bsLarge);
	ButtonType
	Special types of buttons allowed:
	TButtonType = (btNone, btDropDown, btSplit, btGallery); // btNone - normal button // btDropdown - button display a dropdown menu // btSplit - button has dropdown menu and a default action // btGallery - button displays a gallery as a dropdown
	GroupPosition
	Position of this command in a group of commands:
	TGroupPosition = (gpNone, gpStart, gpMiddle, gpEnd, gpSingle); // Determines whether a border is drawn around a button // gpNone - no border drawn // gpStart - the right edge of a border isn't drawn // gpMiddle - top and bottom of border is drawn // gpEnd - the left edge isn't drawn // gpSingle - the full border is drawn

csCheckBox	Command appears like an Office 2007 checkbox.

csComboBox	AllowResize Controls whether the popup menu associated with the combo box can be resized:
	TGalleryResize = (grNone, grVertical, grBoth); // grNone – don't allow resizing // grVertical – only allow vertical resizing // grBoth – allow both vertical and horizontal resizing
	Items
	Items to display when the user selects the dropdown button. These items display in a window like a regular combobox.
	Text
	The text to display in the combobox.
	Width
	The width of the spinner control, excluding any caption width.

csControl	ContainedControl Reference to the control it is associated with on the ribbon.
csCustom	User-defined.

csGallery	The csGallery CommandProperties class descends from the csButton CommandProperties class. This means all of the csButton properties are also available for csGallery commands, as well as the following gallery ones.
	GalleryType
	Type of gallery to be displayed:
	TGalleryType = (gtDropDown, gtGrid, gtRibbon); // gtDropDown - Gallery is in the form of a dropdown menu // gtGrid - Gallery is a grid layout. Meaning it can be more than one item across. ItemsPerRow controls the number of items across // gtRibbon - Gallery is in Ribbon
	ItemsPerRow
	Number of items to display on a row.
	ShowRichContent
	Set to true to display rich content for the gallery items.

csMenu	ShowRichContent: Set to true to display rich content for the menu item. Content: The content to be displayed as rich content. Font: Use a custom font when displaying the rich content. Rich content menu items are limited to two lines. Depending on the content, the menu items might be wider and are not changed to be taller, and some menu items therefore display additional lines (beyond the two permitted.)
csRadioButton	Command appears like an Office 2007 radio button.

csSeparator	csText and csSeparator share the same CommandProps class, so the available properties are the same. Alignment: The alignment of the caption text. EllipsisPosition: Position of any ellipsis if caption text is too wide for the menu width. It is only possible for menu items to be displayed in the Application menu. Font: Font to use when drawing the caption text. Width: Force the width of the created control to be a specific size. Leave as -1 for csSeparator CommandStyle and modify as required for csText CommandStyle.
csText	csText and csSeparator share the same CommandProps class, so the available properties are the same.

Additional Ribbon Component Properties

KeyTip: The key that can be pressed to activate the command when using the keyboard. To activate KeyTips, press either the Alt or F10 key.

NewCol: Setting NewCol for a command forces the control to be displayed in a new column of the ribbon group. This property is only effective when the ribbon group alignment is gaVertical.

NewRow: Setting NewRow for a command forces the control to be displayed in a new row of the ribbon group. This property is only effective when the ribbon group alignment is gaHorizontal.

Default: Set to true to force a menu items caption to be bolded. Only effective for menu items.

See Also

VCL Overview (see page 36)

Building VCL Forms Applications With Graphics (see page 93)

Creating an Application that Uses Ribbon Controls (see page 143)

2.6 WebSnap Procedures

This section provides how-to information on developing WebSnap applications.

Please note that WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

Topics

Name	Description
Building a WebSnap Application (see page 148)	The following procedure describes the generic steps required to build a simple WebSnap project. For more advanced topics, refer to related information following the procedure. Building a WebSnap application consists of five major steps: 1. Create an WebSnap project. 2. Change included components (optional). 3. Set page options (optional) 4. Create additional WebSnap pages. 5. Run the application. **Note:** WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation... more (see page 148)
Building a WebSnap "Hello World" Application (see page 149)	Though simple, the WebSnap "Hello world" application demonstrates the essential steps for creating an WebSnap application. Building the WebSnap "Hello world" application consists of five major steps: 1. Create a WebSnap project. 2. Accept the default included components. 3. Set the page title in the page options. 4. Modify the HTML template. 5. Run the application.
Debugging a WebSnap Application using the Web Application Debugger (see page 150)	This topic describes the essential tasks for debugging a WebSnap application using the Web Application Debugger.
Using the HTML Tag Editor (see page 151)	When you are creating or editing an HTML file, you can use the **Tag Editor** window, beneath the Form Designer, to edit the HTML tags. If you are using an HTML Form, you can display the Tag Editor in the Designer by selecting **View ▶ Tag Editor**. The **Tag Editor** lets you review and modify HTML tags while viewing the corresponding controls in the Designer window, above it. The **Tag Editor** allows you to use the **Code Completion**, **Error Insight**, and **Live Template Completion** features that are also available in the **Code Editor**. Refer to the links... more (see page 151)

2.6.1 Building a WebSnap Application

The following procedure describes the generic steps required to build a simple WebSnap project. For more advanced topics, refer to related information following the procedure.

Building a WebSnap application consists of five major steps:

1. Create an WebSnap project.
2. Change included components (optional).
3. Set page options (optional)
4. Create additional WebSnap pages.
5. Run the application.

Note: WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

To create a WebSnap project

1. Choose **File** ▸ **New** ▸ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **WebSnap Application** from the **Delphi Projects** ▸ **WebSnap** folder.
3. Click **OK**. The **New WebSnap Application** dialog appears.
4. Select the type of application you are creating.
5. Select your application model components.
6. In the **Page Name** field, enter the name of your page.
7. Select your caching type from the **Caching** drop-down.

To change included components (optional)

1. In the **New WebSnap Application** dialog, click **Components**. The **WebApp Components** dialog appears.
2. Select the components you want to include.

   Tip: In most cases, the default settings will suffice.

3. Click **OK**.

To set page options (optional)

1. In the **New WebSnap Application** dialog, click **Page Options**. The **WebApp Components** dialog appears.
2. Set the page options.

To create additional WebSnap pages

1. In the **New Items** dialog, select **WebSnap Page Module** from the **Delphi Projects** ▸ **WebSnap** folder.
2. Configure the page module options and click **OK**.
3. Add and configure components.

See Also

Web Applications Overview (see page 41)

Building an WebSnap "Hello world" Application (see page 149)

2.6.2 Building a WebSnap "Hello World" Application

Though simple, the WebSnap "Hello world" application demonstrates the essential steps for creating an WebSnap application.

Building the WebSnap "Hello world" application consists of five major steps:

1. Create a WebSnap project.
2. Accept the default included components.
3. Set the page title in the page options.
4. Modify the HTML template.
5. Run the application.

To create a WebSnap project

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog appears.
2. In the **New Items** dialog, select **WebSnap Application** from the **Delphi Projects** ▶ **WebSnap** or **C++Builder Projects** ▶ **WebSnap** folder.
3. Click **OK**. The **New WebSnap Application** dialog appears.
4. Select the **Web App Debugger executable** radio button.
5. In the **Class Name** field, enter `HelloWorld`.
6. Select your application model components.
7. In the **Page Name** field, enter `HelloWorld`.
8. Select your caching type from the **Caching** drop-down.

To change included components (optional)

1. In the **New WebSnap Application** dialog, click **Components**. The **WebApp Components** dialog appears.
2. Select the components you want to include.

   Tip: In most cases, the default settings will suffice.

3. Click **OK**.

To set the page title in the page options

1. In the **New WebSnap Application** dialog, click **Page Options**. The **WebApp Components** dialog appears.
2. In the **Title** field, enter `Hello World!`.

To modify the HTML template

1. Click on the HTML tab in the IDE.
2. Below the line `<h2><%= Page.Title %></h2>`, insert a line saying `This is my first WebSnap application.`
3. Save the application.

To run the "Hello world" application

1. Choose **Run** ▶ **Run**. An application window opens, and the COM server registers your WebSnap application with the Web Application Debugger.

2. Close the application window.
3. Choose **Tools** ▸ **Web App Debugger** . The Web Application Debugger launches.
4. In the Web App Debugger, click the **Start** button.
5. Click on the **Default URL** to launch the browser.
6. In the browser, select your Hello World application from the list of applications and click **Go**. Your application appears in the browser with the text `Hello World! This is my first WebSnap application.`
7. Close the Web browser to return to the IDE.

Note: WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

See Also

Web Applications Overview (see page 41)

Building a WebSnap Application (see page 148)

2.6.3 Debugging a WebSnap Application using the Web Application Debugger

This topic describes the essential tasks for debugging a WebSnap application using the Web Application Debugger.

To debug a WebSnap Application using the Web Application Debugger

1. Register the server information application for the Web Application Debugger.
2. Register your WebSnap application with the Web Application Debugger the first time you run it.
3. Launch the Web Application Debugger.
4. Select and launch your web application.
5. Debug your web application using breakpoints and the Web Application Debugger log.

To register the server information application for the Web Application Debugger

1. Navigate to the `bin` directory of your RAD Studio installation.
2. Run `serverinfo.exe`.
3. Close the blank application window that opens.

This step only needs to be performed the first time you use the Web Application Debugger.

To register your web application with the Debugger

1. Choose **Run** ▸ **Run**. This displays the console window of the COM server that is your Web server application.
2. Close the blank application window that opens.

Your COM server is now registered so that the Web App debugger can access it.

To launch the Web Application Debugger

1. Choose **Tools** ▸ **Web App Debugger** . The Web Application Debugger launches.
2. In the Web App Debugger, click the **Start** button.
3. Click on the **Default URL** to launch the browser.

To select and launch your web application

1. In the browser, select your application from the list of applications.
2. Click **Go**. Your application appears in the browser.

   Note: WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

See Also

Web Applications Overview (see page 41)

Building an WebSnap Application (see page 148)

2.6.4 Using the HTML Tag Editor

When you are creating or editing an HTML file, you can use the **Tag Editor** window, beneath the Form Designer, to edit the HTML tags. If you are using an HTML Form, you can display the Tag Editor in the Designer by selecting **View** ▸ **Tag Editor**.

The **Tag Editor** lets you review and modify HTML tags while viewing the corresponding controls in the Designer window, above it. The **Tag Editor** allows you to use the **Code Completion**, **Error Insight**, and **Live Template Completion** features that are also available in the **Code Editor**. Refer to the links at the end of this topic for more information about using each of these features.

The **Tag Editor** works with one tag at a time, unless you have the Document object selected or you have zoomed out from a tag. (When the document object is selected, you'll see the item "DOCUMENT" on the Object Inspector.)

Use the zoom buttons to zoom out to a tag's parent and zoom back in to the selected child tag. Zooming isn't specific to the tag, it's more generic to the markup in the document itself. For example, if the cursor is on a tag in your HTML markup, and you use the Zoom command, it will take you to the outer tag, or one level above the attribute where the cursor is positioned.

Validation against standard HTML style rules occurs automatically. If validation fails, the incorrect element is highlighted in red in the **Designer**, and **Error Insight** appears in the **Tag Editor** to help you correct the problem.

To view HTML code for an individual control

1. With the Designer displayed, drag an HTML element from the **Tool Palette** to the Designer surface. The **Tag Editor** displays the HTML code.
2. To view the individual control's code, click anywhere on the Designer surface to deselect the control. The HTML code appears in the tag editor window, with syntax highlighting. The gray header of the tag editor now displays the higher level tag, usually the FORM tag that defines this particular Web Form.

   Note: If a control is defined using several lines of HTML code, when you select the control, the first line of the code is displayed in the gray header of the tag editor. The additional code appears below in the tag editor window.

To view the HTML code for all controls

1. With the Designer displayed, drag several HTML elements from the **Tool Palette** to the Designer surface. The editor displays the HTML code for each element as you drop them on the Designer surface.
2. Click anywhere on the Designer surface to deselect all controls. This displays the code for all the controls in the tag editor, with syntax highlighting.

To modify a control

1. Click anywhere on the Designer surface to deselect all controls.
2. Locate the tag that corresponds to the control you want to modify.
3. Modify the code, and the change is immediately reflected in the control on the Designer surface.
4. Save your project to make the modifications permanent.

To change editor properties

1. Choose **Tools** ▸ **Options** ▸ **HTMLOptions**.
2. Change any code editor properties.
3. Click **OK**. Your changes take effect immediately.

To zoom between contents of the form and the form container

1. To zoom out so that you can view the HTML form definition, click the left-hand blue arrow in the gray header of the tag editor.

   Note: You can only use this feature when the cursor is somewhere in the tag editor, rather than on the Designer surface.

2. To zoom in so that you can view only the content within the FORM tags, click the right-hand blue arrow in the gray header of the tag editor.

   Note: You can only use this feature when the cursor is somewhere in the tag editor, rather than on the Designer surface.

To close the Tag Editor

1. Choose **Tools** ▸ **Options** ▸ **HTML Options**.
2. Uncheck the **Display Tag Editor** option.
3. Click **OK**.

See Also

Using the Code Editor

Customizing the Code Editor

Using Live Templates

Using Code Insight

2.7 Web Services Procedure

This section provides how-to information on developing and using web services.

Topics

Name	Description
Building a "Hello World" Web Services Application (see page 153)	Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services use SOAP, a standard lightweight protocol for exchanging information in a distributed environment. It uses HTTP as a communications protocol and XML to encode remote procedure calls.

2.7.1 Building a "Hello World" Web Services Application

Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services use SOAP, a standard lightweight protocol for exchanging information in a distributed environment. It uses HTTP as a communications protocol and XML to encode remote procedure calls.

To build a "Hello World" Web Services application

1. Choose **File** ▶ **New** ▶ **Other**. The **New Items** dialog box appears.
2. Select **WebServices** folder.
3. Double-click the **Soap Server Application** icon. The **SOAP Server Application** wizard opens.
4. Choose the type of Web server application you want to use for your Web Service. The wizard generates a new Web server application that includes a Web module that contains three components—HTTPSoapPascalInvoker, HTTPSoapDispatcher, and WSDLHTMLPublish. When you exit the SOAP Server Application wizard, it asks you if you want to define an interface for your Web Service.
5. To create a a Web Service from scratch, click **Yes**. The **Add New Web Service** wizard opens.
6. To add a new Web Service, specify the name of the invokable interface you want to expose to clients. The **Add New Web Service** wizard lets you specify the name of the invokable interface and generates the code to declare and register the interface and its implementation class.
7. To implement a Web Service that has already been defined in a WSDL document, use the WSDL importer to generate the interfaces

To use the WSDL importer

1. Choose **File** ▶ **New** ▶ **Other**. A **New Items** dialog appears.
2. Select **WebServices** folder.
3. Double-click the icon labeled **WSDL importer**. A **WSDL Import Wizard** appears.
4. In that dialog box, either specify the file name of a WSDL document (or XML file) or click the ellipsis button [...] to browse for the file.
5. If the WSDL document is on a server that requires authentication, click **Options**. On the **Import Options** dialog box, enter the user name and password required before the wizard can retrieve the WSDL document. After completing the import options, click **OK**.
6. Click **Next**. The WSDL importer displays the code it generates for every definition in the WSDL document that is compatible with the Web Services framework.

7. Click **Finish**. The importer creates new units that define and register invokable interfaces for the operations defined in the document, and that define and register remotable classes for the types that the document defines.

See Also

Web Services Overview (see page 44)

Using Web Services (see page 2291)

Using Web Services (chapter index) (see page 2289)

WSDL Import Wizard

3 Reference

Topics

Name	Description
C++ Reference (see page 156)	This section contains reference topics for the C++ library in RAD Studio.
Win32 Developer's Guide (see page 1217)	This section contains the Win32 Developer's Guide topics for the Delphi Win32 personality in RAD Studio.

3.1 C++ Reference

This section contains reference topics for the C++ library in RAD Studio.

Topics

Name	Description
Command Line Utilities (see page 156)	C++Builder provides a rich set of command line utilities in addition to its integrated development environment (IDE). These utilities enable you to use the command line to perform targeted compilation and other related functions, including file search and import of definition files.
C++ Compiler Errors And Warnings (C++) (see page 216)	This section describes the RAD Studio C++ compiler error and warning messages.
C++ Language Guide (see page 385)	This sections contains C++ language topics.
C Runtime Library Reference (see page 711)	RAD Studio has several hundred functions, macros, and classes that you call from within your C and C++ programs to perform a wide variety of tasks, including low- and high-level I/O, string and file manipulation, memory allocation, process control, data conversion, mathematical calculations, and more. **Note:** In the online help, each function, macro, and class in the C Runtime Library is listed only once . However, some functions, macros, and classes are defined in more than one header file. For example, `_strerror` is defined in both string.h and stdio.h. For functions that are defined in several header files, the online... more (see page 711)

3.1.1 Command Line Utilities

C++Builder provides a rich set of command line utilities in addition to its integrated development environment (IDE). These utilities enable you to use the command line to perform targeted compilation and other related functions, including file search and import of definition files.

Topics

Name	Description
BCC32, the C++ Command-Line Compiler (see page 159)	The CodeGear C++ compiler (BCC32.EXE) is a resource compiler shell. It invokes BRCC32 and RLINK32, depending on the command-line syntax.
BRC32, the Resource Shell (see page 163)	The Borland resource compiler (BRC32) is a resource compiler shell. It invokes BRCC32 and RLINK32, depending on the command-line syntax.
BRCC32.EXE, the Resource Compiler (see page 165)	BRCC32 is the command-line version of the resource compiler. It accepts a resource script file (.RC) as input and produces a resource object file (.RES) as output. RAD Studio provides a choice in resource compilers. You can choose to use either BRCC32 or RC (the Microsoft SDK resource compiler) on the **Project** ▶ **Options** ▶ **Resource Compiler** dialog box.
COFF2OMF.EXE, the Import Library Conversion Tool (see page 166)	COFF2OMF converts a COFF import library file (InputFile) to a corresponding OMF import library file (OutputFile). COFF2OMF.EXE is located in the C++Builder \bin directory.
CPP32.EXE, the C Compiler Preprocessor (see page 167)	CPP32.EXE produces a file that lists a C or C++ program, in which all #include files and #define macros have been expanded. While you do not need to use the preprocessor during normal compilation, you may find the list file helpful for debugging purposes. Often, when the compiler reports an error inside a macro or an include file, you can get more information about what the error is if you can see the include files or the results of the macro expansions. In many multi-pass compilers, a separate pass performs this work, and the results of the pass can be... more (see page 167)
DCC32.EXE, the Delphi Command Line Compiler (see page 169)	DCC32 is the Delphi (Object Pascal) command line compiler. To display command line help, enter:

GREP.EXE, the text search utility (see page 170)	GREP (Global Regular Expression Print) is a powerful text-search program derived from the UNIX utility of the same name. GREP searches for a text pattern in one or more files or in its standard input stream.
ILINK32.EXE, the Incremental Linker (see page 174)	ILINK32 links object modules (.OBJ files), library modules (.LIB files), and resources to produce executable files (.EXE, .DLL, and .BPL files). ILINK32 creates and maintains a series of state files that contains this information. These state files allow subsequent links to be incremental, greatly reducing the total link time.
IMPDEF.EXE, the Module Definition Manager (see page 179)	Import libraries provide access to the functions in a Windows DLL. Import libraries contain records. Each record contains the name of a DLL and specifies where in the DLL the imported functions reside. These records are bound to the application by the linker and provide Windows with the information necessary to resolve DLL function calls. You can substitute an import library for part or all of the IMPORTS section of a module definition file. IMPDEF takes as input a DLL name, and produces as output a module definition file with an EXPORTS section containing the names of functions exported by... more (see page 179)
IMPLIB.EXE, the Import Library Tool (see page 181)	IMPLIB takes as input either DLLs or module definition files, or both, and produces an import library as output. If you've created a Windows application, you've already used IMPORT32.LIB, the import library for the standard Windows DLLs. IMPORT32.LIB is linked automatically when you build a Win32 application in the C++Builder IDE and when using BCC32 at the command line. An import library lists some or all of the exported functions for one or more DLLs. IMPLIB creates an import library directly from DLLs or from module definition files for DLLs (or a combination of the two).
Using Include Files (see page 182)	In C++, include files always have the file extension .h.
MAKE (see page 183)	MAKE.EXE is a command-line utility that helps you manage project compilation and link cycles. MAKE is not inherently tied to compiling and linking, but is a more generic tool for executing commands based on file dependencies. MAKE helps you quickly build projects by compiling only the files you have modified since the last compilation. In addition, you can set up rules that specify how MAKE should deal with the special circumstances in your builds.
MAKE Directives (see page 186)	MAKE directives resemble directives in languages such as C and Pascal. In MAKE, directives perform various control functions, such as displaying commands onscreen before executing them. MAKE directives begin either with an exclamation point or a period, and they override any options given on the command line. Directives that begin with an exclamation point must appear at the start of a new line. The following table lists the MAKE directives and their corresponding command-line options:
MAKE Macros (see page 191)	A macro is a variable that MAKE expands into a string whenever MAKE encounters the macro in a makefile. For example, you can define a macro called LIBNAME that represents the string "mylib.lib." To do this, type the line LIBNAME = mylib.lib at the beginning of your makefile. Then, when MAKE encounters the macro $(LIBNAME), it substitutes the string mylib.lib. Macros let you create template makefiles that you can change to suit different projects. To use a macro in a makefile, type $(MacroName) where MacroName is a defined macro. You can use either braces or parentheses to enclose MacroName. MAKE... more (see page 191)
MAKE Rules (Explicit and Implicit) and Commands (see page 193)	You write explicit and implicit rules to instruct MAKE how to build the targets in your makefile. In general, these rules are defined as follows: • Explicit rules are instructions for specific files. • Implicit rules are general instructions for files without explicit rules. All the rules you write follow this general format:
Message Options (see page 197)	Use the -w option to specify message options for the CodeGear C++ compiler: • To enable a particular warning message, enter the -w option with a one- to three-letter option code. • To disable the warning message, enter the -w- option with a one- to three- letter option code.

Module Definition Files (see page 199)	You use module definition files with ILINK32. A module definition file is an ASCII text file that provides information to ILINK32 about the contents and system requirements of a Windows application. Use IMPDEF to create a module definition file. The module definition file names the .EXE or .DLL, identifies the application type, lists imported and exported functions, describes the code section and data segment attributes, lets you specify attributes for additional code sections and data segments, specifies the size of the stack, and provides for the inclusion of a stub program.
Using Precompiled Header Files (see page 203)	Precompiled header files can dramatically increase compilation speed by storing an image of the symbol table on disk in a file, then later reloading that file from disk instead of parsing all the header files again. Directly loading the symbol table from disk is much faster than parsing the text of header files, especially if several source files include the same header file. To use precompiled header files, specify the various -H options in your BCC32 command. ***Precompiled Header Options***
RLINK32.DLL, the Resource Linker (C++) (see page 204)	RLINK32.DLL is the resource linker that binds resources, in .RES file form, to an .EXE file, and marks the resulting .EXE file as a Windows executable. RLINK32.DLL also: • Links the resources by fixing up string tables and message tables and then binding these linked resources into the executable. • Is called by ILINK32 and is used for 32-bit resources.
TDUMP.EXE, the File Dumping Utility (see page 204)	TDUMP.EXE produces a file dump that shows the structure of a file. TDUMP breaks apart a file structurally and uses the file's extension to determine the output display format. TDUMP recognizes many file formats, including .EXE, .OBJ, and .LIB files. If TDUMP doesn't recognize an extension, it produces a hexadecimal dump of the file. You can control the output format by using the TDUMP command-line options when you start the program. TDUMP's ability to peek at a file's inner structure displays not only a file's contents, but also how a file is constructed. Moreover, because TDUMP verifies that a file's... more (see page 204)
TLIB.EXE, the Library Manager (see page 208)	TLIB is a utility that manages libraries of .OBJ (object module) files. A library is a convenient way to deal with a collection of object modules as a unit. The libraries included with the CodeGear C++ compiler were built with TLIB. You can use TLIB to build your own libraries, or to modify the CodeGear C++ libraries, your libraries, libraries furnished by other programmers, or commercial libraries you've purchased. When TLIB modifies an existing library, TLIB creates a copy of the original library and gives it a .BAK extension. You can use TLIB to: • Create a new library from a... more (see page 208)
Using TOUCH.EXE (see page 212)	TOUCH.EXE updates a file's date stamp so that it reflects your system's current time and date.
TRIGRAPH (see page 213)	Trigraphs are three-character sequences that replace certain characters used in the C language that are not available on some keyboards. Translating trigraphs in the compiler would slow compilation down considerably, so CodeGear C++ provides a filter named TRIGRAPH.EXE to handle trigraph sequences.
RC.EXE, the Microsoft SDK Resource Compiler (see page 213)	RC is the command-line version of the standard Microsoft SDK resource compiler. It accepts a resource script file (.RC) as input and produces a resource object file (.RES) as output. Both C++Builder 2009 and Delphi 2009 give you a choice of resource compilers. On the **Project** ▶ **Options** ▶ **Resource Compiler** dialog box, you can select either of the following: • BRCC32.exe, the CodeGear resource compiler • RC.exe, the Microsoft platform SDK Resource Compiler RC supports Unicode characters in resource files and file names, as well as new Vista resource types such as icons with alpha channel. The actual filename of the RC compiler... more (see page 213)

WSDLIMP.EXE, the Command Line WSDL Import Tool (see page 214)	WSDLIMP generates code to represent the types and APIs that a WSDL document defines. This code can be used to write client applications that call on the Web Service that the WSDL (Web Services Description Language) document describes. If you want to write a server that implements the Web Service, one of the command line options tells the importer to generate implementation classes that you can then complete by filling in the bodies of the generated methods. **Note:** WSDLIMP ignores any definitions in the WSDL document for which it can't generate code. That is, it can only import Web Services... more (see page 214)

3.1.1.1 BCC32, the C++ Command-Line Compiler

The CodeGear C++ compiler (BCC32.EXE) is a resource compiler shell. It invokes BRCC32 and RLINK32, depending on the command-line syntax.

Command Line Syntax

```
bcc32 [option [option...}] <filename> [<filename>...]
```

Use spaces to separate the command-line compiler name, each option, and the filenames. Precede each option by either a hyphen (-) or a forward slash (/). For example:

```
BCC32 -Ic:\code\hfiles
```

You can also specify options in configuration (.CFG) files, which are described in a following section.

You can use BCC32 to send .OBJ files to ILINK32 or .ASM files to TASM32 (if you have TASM32 installed on your machine).

See the BCC32 Command Line Help for Detailed Information

To display the BCC32.exe command line help in the cmd window, include the –h command line option.

For example, to display a list of the commonly used compiler command line options, type:

```
BCC32 -h
```

The displayed list indicates the options that are enabled by default (*):

```
C:\>bcc32 -h CodeGear C++ 5.92 for Win32 Copyright (c) 1993, 2007 CodeGear
Available options (* = default setting, xxx = has sub-options: use -h -X):
(Note: -X- or -w-XXX will usually undo whatever was set or unset by -X)
  -3      Generate 80386 protected-mode compatible instructions
  -4      Generate 80386/80486 protected-mode compatible instructions
  -5      Generate Pentium instructions
  -6      Generate Pentium Pro instructions
  -Axxx   Enable ANSI conformance
  -B      Compile to .ASM (-S), then assemble to .OBJ
  -C      Enable nested comments
  -CP     Enable code paging (for MBCS)
  -D      -D <name> defines 'name' as a null string, or use -D<name>=<value>
  -E      Specify which assembler to use
  -G      Optimize for size/speed; use -O1 and -O2 instead
  -Hxxx   Generate and use precompiled headers
  -I      Set the include file search path
  -Jxxx   Template generation options
  -K      Set default character type to unsigned
  -L      Library file search path
  -M      Create a linker map file
  -O      Optimize jumps
  -P      Perform C++ compile regardless of source extension
  -Q      Extended compiler error information
  -R      Include browser information in generated .OBJ files
  -RF     Find references to symbol
* -RT     Enable runtime type information
  -S      Compile to assembly
  -T      Specify assembler option, e.g. -Tx
```

```
  -U      Undefine any previous definitions of name
  -Vxxx   Compatibility options
  -Wxxx   Target is a Windows application
  -X      Disable compiler autodependency output
  -axxx   Set data alignment boundary.  Default is -a8; -a- means -a1
* -b      Make enums integer-sized (-b- makes them short as possible)
  -c      Compile to object file only, do not link
  -d      Merge duplicate strings
  -dc     Put strings into the read-only data segment
  -dw     Put strings into the (writeable) data segment
  -e      Specify target executable pathname
* -ff     Fast floating point
  -fp     Correct Pentium FDIV flaw
* -fq     Use quiet floating point compare instruction (FUCOMP)
  -g      Stop batch compilation after n warnings (Default = 255)
  -h      Request help ('-h -' shows all help). Can be specific: -h -V
  -i      Set maximum significant identifier length (Default = 250)
  -j      Stop batch compilation after n errors (Default = None)
* -k      Generate standard stack frames
  -l      Pass options to the linker; example: -ls -l-x
  -m      Generate makefile dependency information
  -md     Put dependency info in .d files, not in the object file
  -mm     Ignore system header files while generating dependency info
  -mo     Specify the output file for dependency info
  -n      Set output directory for object files
  -o      Set output filename (-o<filename> or –o <filename> supported)
  -pxxx   Use Pascal calling convention
  -q      Suppress compiler identification banner
  -r      Use register variables
  -rd     Use register variables only when register keyword is employed
  -s      Link using the system's non-incremental linker
  -txxx   An alternate name for the -Wxxx switches; there is no difference
* -u      Generate underscores on symbol names
  -vxxx   Turn on source debugging
  -w      Display all warnings
  -w!     Return non-zero from compiler on warnings
  -xxxx   Enable exception handling
  -y      Debug line numbers on
  -z      Options for redefining standard segment names
```

Displaying Help for Specific Options, Groups such as -Axxx and -Vxxx

You can get more specific information about each of the multi-letter options, such as `-Axxx` (language compatibility and standards compliance) and `-Vxxx` (backward compatibility).

To do this, use the `-h` command line option with the initial letter of the option group (such as `-A` to specify the `-Axxx` options). BCC32 will display only the help topics for the specified set of options (such as –Axxx, –Vxxx or –Wxxx).

For example, to display a description of the -Axxx (language compatibility and standards compliance) options, use the –h and –A command line options:

```
C:\>bcc32 -h -A
CodeGear C++ 6.10 for Win32 Copyright (c) 1993-2008 CodeGear
Available options (* = default setting, xxx = has sub-options: use -h -X):
(Note: -X- or -w-XXX will usually undo whatever was set or unset by -X)
  -A      Enable ANSI conformance
  -AF     Use SUN Forte keywords and extensions
  -AG     Use GNU keywords and extensions
  -AK     Use Kernighan and Ritchie keywords and extensions
  -AT     Use CodeGear C++ keywords and extensions (also -A-)
  -AU     Use UNIX System V keywords and extensions
  -An     Use C99 keywords and extensions
  -Ax     Use C++-0x keywords and extensions
```

In the following example, the BCC32 command line help displays details about all the `-Vxxx` (backward compatibility) options:

```
C:\>bcc32 -h -V
CodeGear C++ 6.10 for Win32 Copyright (c) 1993-2008 CodeGear
Available options (* = default setting, xxx = has sub-options: use -h -X):
(Note: -X- or -w-XXX will usually undo whatever was set or unset by -X)
  -V      Compatibility options
  -V0     External C++ virtual tables
  -V1     Public C++ virtual tables
* -VA     Generate all global functions in their own virtual/weak segment
  -VC     Do not mangle calling convention into symbols
  -VF     MFC compatibility
  -VF3    Support MFC 3.2
  -VF4    Support MFC 4.0
* -VI     Use Microsoft search algorithm to locate header files
  -VM     Microsoft Visual C++ compatibility
  -Va     Support old-style class arguments
  -Vb     Enable backward compatability with Bcc versions 5.8.2 and earlier
  -Vbc    Don't collapse reference to reference and allow qualified references
  -Vbe    Allow old-style explicit template specialization
  -Vbn    Allow calling of non-const or non-volatile member function for a const
 or volatile object
  -Vbo    Use old Borland overload resolution rules
  -Vbr    Allow non-const reference binding
  -Vbs    Treat string literals as non-const
  -Vbt    Use old Borland type rules for ternary operators
  -Vbx    Allow explicit template specialization as a member function
  -Vc     Support constructor displacements
  -Vd     Use old C++ for-statement scoping rules
  -Ve     Zero-length empty base classes
  -Vg     Disable lexical digraph scanner
  -Vi     Use old 8.3 search algorithm to locate header files
  -Vl     Use old Borland class layout
  -Vm     Member pointer options
  -Vmd    Use the smallest possible representation for member pointers
  -Vmm    Support multiple inheritance for member pointers
  -Vmp    Honor declared precision of member pointers
  -Vms    Support single inheritance for member pointers
  -Vmv    Place no restrictions on where member pointers can point
  -Vn     Enable new operator names: and, or, and_eq, bitand, etc.
  -Vo     Set (almost) all compatibility flags; used with old code
  -Vp     Push 'this' first, as does Pascal
  -Vr     Reverse order for Multi-character constant
  -Vs     Use old-style virdef generation
  -Vt     Put virtual table pointer at front of object layout
  -Vv     Use 'slow' virtual base pointers
  -Vw     Emit native code instead of Unicode for multi-byte character
  -Vx     Zero-length empty class member functions
```

Default Settings

BCC32.EXE has specific options that are on by default. To turn off a default option or to override options in a configuration file, follow the option with a minus (-) sign.

Files with the .CPP extension compile as C++ files. Files with a .C extension, with no extension, or with extensions other than .CPP, .OBJ, .LIB, or .ASM compile as C files.

The compiler tries to link with a module-definition file with the same name as the executable, and extension .DEF.

General Compiler Output Options

Option	Description	Details
-c	Compiles to .OBJ, no link	Compiles and assembles the named .C, .CPP, and .ASM files, but does not execute a link on the resulting .OBJ files.
-e <filename>	Specify executable filename	Link file using <filename> as the name of the executable file. If you do not specify an executable name with this option, the linker creates an executable file based on the name of the first source file or object file listed in the command.
-l <x>	Pass option to linker.	Use this command-line option to pass option(s) <x> to the linker from a compile command. Use the command-line option `-l-x` to disable a specific linker option.
-M	Create a MAP file	Use this compiler option to instruct the linker to create a map file.
-o <filename>	Compike .OBJ to <filename>	Compiles the specified source file to <filename>.OBJ
-P	C++ compile	Causes the compiler to compile all source files as C++ files, regardless of their extension. Use -P- to compile all .CPP files as C++ source files and all other files as C source files. The command-line option -Pext causes the compiler to compile all source files as C++ files and it changes the default extension to whatever you specify with ext. This option is provided because some programmers use different extensions as their default extension for C++ code. The option -P-ext compiles files based on their extension (.CPP compiles to C++, all other extensions compile to C) and sets the default extension (other than .CPP).
-tWM	Generate a multi-threaded target	Creates a multi-threaded .EXE or .DLL. This option is not needed if you include a module definition file (.DEF file) in your compile and link commands which specify the type of 32-bit application you intend to build.

Compiler option precedence rules

The command-line compilers evaluate options from left to right, and follow these rules:

- If you duplicate any option (except for the options -D, -I, -L, or -U), the last option typed overrides any earlier one.
- Options typed at the command line override configuration and response file options except for the -D, -I, -L, and -U options, which are cumulative.

Specifying directories in command line options

The CodeGear C++ compiler can search multiple directories for include and library files. The syntax for the library directories option (-L) and the include directories option (-I), (like the #define option (-D)) allows multiple listings of a given option. Here is the syntax for these options:

```
-L <dirname> [<dirname>;...]
-I <dirname> [<dirname>;...]
```

The parameter <dirname> used with -L and -I can be any directory or directory path. You can enter these multiple directories on the command line in the following ways

- You can stack multiple entries with a single -L or -I option by using a semicolon: `BCC32.EXE –L dirname1;dirname2;dirname3 –I include1;include2;include3 myfile.c`
- You can place more than one of each option on the command line, like this: `BCC32.EXE –L dirname1 –L dirname2 –L dirname3 –I include1 –I include2 –I include3 myfile.c`
- You can mix styles: `BCC32.EXE –L dirname1;dirname2 –Ldirname3 –I include1;include2 –I include3`

```
myfile.c
```

If you list multiple -L or -I options on the command line, the result is cumulative. The compiler searches all the directories listed, in order from left to right.

Using compiler configuration files (.CFG files)

If you repeatedly use a certain set of options, you can list them in a configuration file instead of continually typing them on the command line. A configuration file is a standard ASCII text file that contains one or more command-line options. Each option must be separated by a space or a new line.

Whenever you issue a compile command, BCC32.EXE searches for a configuration file called BCC32.CFG. The compiler looks for the .CFG file first in the directory where you issue the compile command, then in the directory where the compiler is located.

You can create and use multiple configuration files in addition to using the default .CFG file.

To use a configuration file, use the following syntax where you would place the compiler options:

```
+[path]filename
```

For example, you could use the following command line to use a configuration file called MYCONFIG.CFG:

```
BCC32 +C:\MYPROJ\MYCONFIG.CFG mycode.cpp
```

Options typed on the command line override settings stored in configuration files except for the prepended options -D, -I, -L, and -U.

Using response files

Response files let you list both compiler options and file names in a single file (unlike configuration files, which accept only compiler options). A response file is a standard ASCII text file that contains one or more command-line options and/or file names, with each entry in the file separated by a space or a new line. In addition to simplifying your compile commands, response files let you issue a longer command line than most operating systems allow.

The syntax for using a single response file is:

```
BCC32 @[path]respfile.txt
```

The syntax for using multiple response files is:

```
BCC32 @[path]respfile.txt @[path]otheresp.txt
```

Response files typically have an .RSP extension.

Options typed at the command line override any option or file name in a response file except for -D, -I, -L, and -U, which are prepended.

See Also

Using Include Files (see page 182)

Using Precompiled Header Files (see page 203)

Using Module Definition Files (see page 199)

RLINK32.DLL (see page 204)

BRCC32.EXE (see page 165)

3.1.1.2 BRC32, the Resource Shell

The Borland resource compiler (BRC32) is a resource compiler shell. It invokes BRCC32 and RLINK32, depending on the command-line syntax.

Command Line Syntax

```
brc32 [options] <filename>.RC [<filename>.EXE]
```

To display command line help, enter:

```
brc32
```

Command Line Options

Option	Description
-d<name> [=string]	Defines a symbol you can test with the #IFDEF preprocessor directive.
-fo<filename>	Renames the .RES file.
-fe<filename>	Renames the .EXE file.
-i<path>	Adds one or more directories (separated by semicolons) to the include search path.
-k	Disables the contiguous preload of segments and resources in the .EXE file. Segments are kept in the order in which they appear in the .DEF file. (This option only applies to 16-bit resources and is disabled when the -r option is in effect.)
-r	Creates a .RES file only. The compiled .RES file is not added to the .EXE.
-v	Prints progress messages (verbose listing).
-x	Directs the compiler to ignore the INCLUDE environment variable when it searches for include or resource files.
-16	Builds 16–bit .RES files.
-32	Builds 32–bit .RES files.
-Vd.d	Makes the .EXE file with Windows version provided (v3.1 is the default for 16-bit resources; -v4.0 is the default for 32-bit resources). Version options are listed in the following table.

Version Options (Used with —Vd.d Option)

Option	Bit	Resulting Look
3.1	16	Gives white background with a non 3-D look for Windows 3.1x, Windows 32s, or WinNT 3.1
4.0	16	Gives gray 3-D look for Windows 95 and WinNT 3.51
3.1	32	Gives white background with a non 3-D look for Windows 32s or WinNT 3.1
4.0	32	Gives gray 3-D look for Windows 95 and WinNT 3.51

Options for Downward Compatibility

Option	Description
-t	Creates an application that runs only in protected mode (Windows Standard or 386 Enhanced mode).
-31	Builds Windows 3.1-compatible .RES files.
-w32	Builds Win32-compatible .RES files.

Resource Shell Examples

The following statement compiles the .RC file, creates a .RES file, and adds the .RES file to the executable file:

```
brc32 <filename>.RC <filename>.EXE
```

BRC32 automatically seeks an .EXE file with the same name as the .RC file. You need to specify the .EXE file only if its name is different from that of the .RC file.

The following statement creates a .RES file, but not an .EXE file. If you name an .EXE file in the command line, BRC ignores it:

```
brc32 -r <filename>.EXE
```

The following statement adds an existing .RES file to an executable file. The .EXE file name is required only if it differs from the .RES file name:

```
brc32 <filename>.RES <filename>.EXE
```

This example uses BRC32 to build a 16-bit Windows 3.1 compatible .RES file:

```
brc32 -16 -v3.1 -fo<filename>.RES <filename>.RC
```

See Also

RLINK32.DLL (see page 204)

BRCC32.EXE (see page 165)

3.1.1.3 BRCC32.EXE, the Resource Compiler

BRCC32 is the command-line version of the resource compiler. It accepts a resource script file (.RC) as input and produces a resource object file (.RES) as output.

RAD Studio provides a choice in resource compilers. You can choose to use either BRCC32 or RC (the Microsoft SDK resource compiler) on the **Project** ▸ **Options** ▸ **Resource Compiler** dialog box.

Command Line Syntax

```
BRCC32 [<options>] <filename>
```

To display command line help, enter:

```
brcc32
```

Or add the help flag:

```
brcc32 -h
```

BRCC32 Command Options

Option	Description
@<responsefile>	Takes instructions from the specified command file.
-c <codepage>	Uses the specified code page for resource translation. If -c is not used, the default ANSI code page is used.
-d<name>[=<string>]	Defines a preprocessor symbol.
-fo<filename>	Renames the output .RES file. (By default, BRCC32 creates the output .RES file with the same name as the input .RC file.)
-i<path>	Adds one or more directories (separated by semicolons) to the include search path.
-l<language>	Specifies default language.
-m	Indicates that the code page specified with the -c switch contains double-byte character set (DBCS) characters.
-r	This switch is ignored. It is included for compatibility with other resource compilers.
-v	Prints progress messages (verbose).
-x	Deletes the current include path.
-h or ?	Displays help.

-16	Builds a 16–bit resource.
-32	Builds a 32–bit resource.

BRCC32 predefines common resource-related Windows constants such as WS_VISIBLE and BS_PUSHBUTTON. Also, two special compiler-related symbols are defined: RC_INVOKED and WORKSHOP_INVOKED. These symbols can be used in the source text in conjunction with conditional preprocessor statements to control compilation.

For example, the following construct can greatly speed up compilation:

```
#ifndef WORKSHOP_INVOKED#include “windows.h” #endif
```

Downward Compatibility

The following syntax and options are supported for downward compatibility:

Option	Description
-16	Builds 16–bit .RES files.
-32	Builds 32–bit .RES files.
-31	Builds Windows 3.1–compatible .RES files.
-w32	Builds Win32–compatible .RES files.

See Also

BRC32.EXE (see page 163)

RC.EXE (see page 213)

3.1.1.4 COFF2OMF.EXE, the Import Library Conversion Tool

COFF2OMF converts a COFF import library file (InputFile) to a corresponding OMF import library file (OutputFile). COFF2OMF.EXE is located in the C++Builder \bin directory.

Command Line Syntax

```
Coff2Omf [<options>] InputFile OutputFile
```

To display command line help, enter:

```
coff2omf
```

COFF2OMF converts the publicly exported symbols in the COFF import library into a corresponding set of symbols in the OMF import library. COFF2OMF enables C++Builder users to link to Microsoft and other third-party DLLs that use the COFF format. COFF2OMF does not convert .OBJ files.

Run the COFF2OMF tool on a COFF import library created for a specific DLL, and use the resulting import library to link OMF format EXEs and DLLs to the COFF DLL.

Command Line Option	Description
-q	Quiet mode. Converts without writing tool and copyright information to standard output.
-v	Dumps selected symbols. Dumps the converted symbols to standard output, which can be redirected to an output file.
-h or -?	Displays help.

-r	Removes (deletes) output file if empty. Deletes the output file if there is an error and the conversion results in an empty file.
-lib:xx	Specifies options for OMF import library generation. The xx options can be: • ms — Allow entries that have MS C++ name mangling. The default is No. • st — Normalize names instead of aliasing MS stdcall mangling. • ca — Don't perform MS cdecl aliasing. Default is to alias.

3.1.1.5 CPP32.EXE, the C Compiler Preprocessor

CPP32.EXE produces a file that lists a C or C++ program, in which all #include files and #define macros have been expanded. While you do not need to use the preprocessor during normal compilation, you may find the list file helpful for debugging purposes.

Often, when the compiler reports an error inside a macro or an include file, you can get more information about what the error is if you can see the include files or the results of the macro expansions. In many multi-pass compilers, a separate pass performs this work, and the results of the pass can be examined. Because the CodeGear C++ compiler is a single-pass compiler, use CPP32 to get the first-pass functionality found

For each file processed by CPP32, the output is written to a file in the current directory (or the output directory named by the -n option) with the same name as the source name but with an extension of .I.

This output file is a text file containing each line of the source file and any include files. Any preprocessing directive lines have been removed, along with any conditional text lines excluded from the compile. Unless you use a command-line option to specify otherwise, text lines are prefixed with the file name and line number of the source or include file the line came from. Within a text line, any macros are replaced with their expansion text. Use the -Sr option to produce a file which doesn't have line numbers. You can then pass this file to the compiler (use the -P compiler option to force a C++ compile).

Command Line Syntax

```
CPP32 [<options>] <filename[s]>
```

To display command line help, enter:

```
cpp32 -h
```

CPP32 recognizes all the same options that BCC32 does, except for the following additions for the -S option (Control preprocessed output format):

Option	Description
-Sc	Keep comments in preprocessed file.
-Sd	Keep defines in preprocessed file.
-Sk	Keep output on errors.
-Sr	Make output readable by preserving comments and indentations
-Ss	Show statistics of file names and line counts.

CPP32 as a Macro Preprocessor

CPP32 can be used as a macro preprocessor; the resulting .i file can then be compiled with BCC32. The following simple program illustrates how CPP32 preprocesses a file.

Source file: HELLOFB.C

```
#define NAME “Frank CodeGear” #define BEGIN { #define END } main() BEGIN
        printf(“%s\n”, NAME);p END
```

CPP32 Command Line

```
CPP32 HELLOFB.C
```

Output (written to HELLOFB.I)

```
/* HELLOFP.C 1: */

/* HELLOFP.C 2: */

/* HELLOFP.C 3: */

/* HELLOFP.C 4: */

/* HELLOFP.C 5: */main()

/* HELLOFP.C 6: */printf("%s\n", "Frank CodeGear");

/* HELLOFP.C 7: */}

/* HELLOFP.C 8: */}
```

Using MIDL with CPP32

MIDL (Microsoft Interface Definition Language) is an RPC compiler. In order to use MIDL with the C++ preprocessor (CPP32.EXE), you must use the following MIDL command:

Option	Description
-<cpp_cmd> {<CPP32>}	Tells MIDL which preprocessor to use when processing an .IDL or .ACF file. MIDL calls the preprocessor to expand macros within source files.
-<cpp_opt> "{<options>}"	Specifies the command line options for the preprocessor. The -Sr option removes line number and file name information from each line of the preprocessed output. The -oCON option indicates that preprocessed output should go to standard output, instead of to file. The preprocessor banner and the current file that is being processed are not emitted. Including -oCON within a .CFG file processed by the preprocessor causes the banner to be emitted.
{<CPP32 options>}	Passes the options to CPP32.
{<MIDL options>}	Any MIDL command-line options.
{<.idl/.acf file>	The source file that MIDL processes.

BCC32, CPP32 and UUIDs

In some cases, CPP32 does not accept valid UUIDs. For example, a valid UUID statement is:

```
uuid(5630EAA0-CA48-1067-B320-00DD010662DB)
```

When CPP32 encounters 5630EAA0, it is classified as a floating-point number, and since it is an invalid floating point number, the preprocessor emits an error. To work around this problem, enclose the UUID within quotes. When using MIDL with CPP32, use the -ms_ext option. The UUID statement becomes:

```
uuid("5630EAA0-CA48-1067-B320-00DD010662DB")
```

and the MIDL command line becomes:

```
MIDL -ms_ext -cpp_cmd CPP32 -cpp_opt " -oCON {<CPP32 options>}" {<MIDL options>} {<.idl/.acf
```

```
file>}
```

See Also

BCC32.EXE (see page 159)

3.1.1.6 DCC32.EXE, the Delphi Command Line Compiler

DCC32 is the Delphi (Object Pascal) command line compiler.

To display command line help, enter:

```
dcc32
```

or:

```
dcc32 –h
```

or:

```
dcc32 –help
C:\>dcc32
CodeGear Delphi for Win32 compiler version 18.5
Copyright (c) 1983,2007 CodeGear

Syntax: dcc32 [options] filename [options]

  -A <unit>=<alias> = Set unit alias
  -B = Build all units
  -CC = Console target
  -CG = GUI target
  -D<syms> = Define conditionals
  -E<path> = EXE/DLL output directory
  -F<offset> = Find error
  -GD = Detailed map file
  -GP = Map file with publics
  -GS = Map file with segments
  -H = Output hint messages
  -I<paths> = Include directories
  -J = Generate C .obj file
  -JP = Generate C++ object file
  -JPN = Generate C++ .obj, include namespace
  -JPE = Generate C++ .obj, export all symbols
  -JPH = Generate C++ .obj, export symbols, generate header .hpp files
  -JPHNE = Generate C++ .obj file, .hpp file, in namespace, export all
  -JL = Generate package .lib, .bpi, and all .hpp files for C++
  -K<addr> = Set image base addr
  -LE<path> = package .bpl output directory
  -LN<path> = package .dcp output directory
  -LU<package> = Use package
  -M = Make modified units
  -N0<package> = Use package
  -M = Make modified units
  -N0<path> = unit .dcu output directory
  -NH<path> = unit .hpp output directory
  -NO<path> = unit .obj output directory
  -NB<path> = unit .bpi output directory
  -NS<namespace> = Namespace search path
  -O<paths>  = Object directories
  -P = look for 8.3 file names also
  -Q = Quiet compile
  -R<paths> = Resource directories
  -U<paths> = Unit directories
  -V = Debug information in EXE
  -VR = Generate remote debug (RSM)
```

```
    -W[+|-][warn_id] = Output warning messages
    -Z = Output 'never build' DCPs
    -$<dir> = Compiler directive
    --help = Show this help screen
    --version = Show name and version
    --codepage:<cp> = specify source file encoding
    --default-namespace:<namespace> = set namespace
    --depends = output unit dependency information
    --doc = output XML documentation
    --drc = output resource string .drc file
    --no-config = do not load default DCC32.CFG file
    --description:<string> = set executable description
  Compiler switches: -$<state> (defaults are shown below)
    A8  Aligned record fields
    B-  Full boolean Evaluation
    C+  Evaluate assertions at runtime
    D+  Debug information
    G+  Use imported data references
    H+  Use long strings by default
    I+  I/O checking
    J-  Writeable structured consts
    L+  Local debug symbols
    M-  Runtime type info
    O+  Optimization
    P+  Open string params
    Q-  Integer overflow checking
    R-  Range checking
    T-  Typed @ operator
    U-  Pentium(tm)-safe divide
    V+  Strict var-strings
    W-  Generate stack frames
    X+  Extended syntax
    Y+  Symbol reference info
    Z1  Minimum size of enum types
```

Creating C++ Files from DCC32

If you want to generate .hpp files (and the corresponding .obj) from a .pas file, you should use -JPHNE.

You can also use -JL on the .dpk file containing the .pas file.

3.1.1.7 GREP.EXE, the text search utility

GREP (Global Regular Expression Print) is a powerful text-search program derived from the UNIX utility of the same name. GREP searches for a text pattern in one or more files or in its standard input stream.

Using GREP

Here is a quick example of a situation where you might want to use GREP. Suppose you wanted to find out which text files in your current directory contained the string "Bob". You would type:

```
grep Bob *.txt
```

GREP responds with a list of the lines in each file (if any) that contained the string "Bob". Because GREP does not ignore case by default, the strings "bob" and "boB" do not match.

GREP can do a lot more than match a single, fixed string. You can make GREP search for any string that matches a particular pattern. (See GREP: The search string.)

Command Line Syntax

The general command-line syntax for GREP is

```
grep [-<options>] <searchstring> [<files(s)>...]
```

To display a list of the GREP command line options, special characters, and defaults for GREP, enter:

```
grep ?
```

GREP Command Line Options

Option	Description
<options>	Consists of one or more letters, preceded by a hyphen (-), that changes the behavior of GREP.
<searchstring>	Gives the pattern to search for.
<file(s)>	Tells GREP which files to search. (If you do not specify a file, GREP searches standard input; this lets you use pipes and redirection.) Files can be an explicit file name or a generic file name incorporating the DOS ? and * wildcards. In addition, you can type a path (drive and directory information). If you list files without a path, GREP searches the current directory. If you do not specify any files, input to GREP must come from redirection (<) or a vertical bar (|).

Redirecting Output from GREP

If you find that the results of your GREP are longer than one screen, you can redirect the output to a file.

For example, you could use this command:

```
GREP "Bob" *.txt > temp.txt
```

This command searches all files with the TXT extension in the current directory, then puts the results in a file called TEMP.TXT. (You can name this file anything you like.) Use any word processor to read TEMP.TXT (the results of the search).

GREP Command Line Options

You can pass options to the GREP utility on the command line by specifying one or more single characters preceded by a hyphen (-). Each individual character is a switch that you can turn on or off: A plus symbol (+) after a character turns the option on; a hyphen (-) after the character turns the option off. The + sign is optional; for example, -r means the same thing as -r+. You can list multiple options individually (like this: -i -d -l), or you can combine them (like this: -ild or -il, -d, and so on).

Here are the GREP option characters and their meanings:

Command Line Options for GREP

Option	Description
-c	Count only: Prints only a count of matching lines. For each file that contains at least one matching line, GREP prints the file name and a count of the number of matching lines. Matching lines are not printed. This option is off by default.
-d	Search subdirectories: For each file specified on the command line, GREP searches for all files that match the file specification, both in the directory specified and in all subdirectories below the specified directory. If you give a file without a path, GREP assumes the files are in the current directory. This option is off by default.
-e	Search expression follows: Indicates that the next argument is the search expression. This option is useful when you want to search for an expression that begins with "-".
-i	Ignore case: GREP ignores upper/lowercase differences. When this option is on, GREP treats all letters a to z as identical to the corresponding letters A to Z in all situations. This option is off by default.
-l	List file names only: Prints only the name of each file containing a match. After GREP finds a match, it prints the file name and processing immediately moves on to the next file. This option is off by default.
-n	Line numbers: Each matching line that GREP prints is preceded by its line number. This option is off by default.

-o	UNIX output format: Changes the output format of matching lines to support more easily the UNIX style of command-line piping. All lines of output are preceded by the name of the file that contained the matching line. This option is off by default.
-r	Regular expression search: The text defined by searchstring is treated as a regular expression instead of as a literal string. This option is on by default. A regular expression is one or more occurrences of one or more characters optionally enclosed in quotes. The following symbols are treated specially: • ^ start of line • . any character • * match zero or more chatacteres • [aeiou0-9] match a, e, i, o, u, and 0-9 • [^aeiou0-9] match all but a, e, i, o, u, and 0-9 • $ end of line • \ quote next character • + match one or more
-u <filename>	Update options: Creates a copy of GREP.EXE called <filename>.EXE. Any options included on the command line are saved as defaults in the new copy of GREP. Use the -u option to customize the default option settings. To verify that the defaults have been set correctly, type filename ?; each option on the help screen is followed by a + or - to indicate its default setting.
-v	Nonmatch: Prints only nonmatching lines. Only lines that do not contain the search string are considered nonmatching lines. This option is off by default.
-w	Word search: Text found that matches the regular expression is considered a match only if the character immediately preceding and following cannot be part of a word. The default word character set includes A to Z, 0 to 9, and the underscore (_). This option is off by default. An alternate form of this option lets you specify the set of legal word characters. Its form is -w[set], where set is any valid regular expression. If you define the set with alphabetic characters, it is automatically defined to contain both the uppercase and lowercase values for each letter in the set (regardless of how it is typed), even if the search is case-sensitive. If you use the -w option in combination with the -u option, the new set of legal characters is saved as the default set.
-z	Verbose: GREP prints the file name of every file searched. Each matching line is preceded by its line number. A count of matching lines in each file is given, even if the count is zero. This option is off by default.
?	Displays a help screen showing the options, special characters, and defaults for GREP.

The Search String

The value of <searchstring> defines the pattern GREP searches for. A search string can be either a regular expression or a literal string.

- In a regular expression, certain characters have special meanings: They are operators that govern the search. (A regular expression is either a single character or a set of characters enclosed in brackets. A concatenation of regular expressions is a regular expression.)
- In a literal string, there are no operators: Each character is treated literally.

You can enclose the search string in quotation marks to prevent spaces and tabs from being treated as delimiters. To search for an expression that begins with "-", use the -e option. The text matched by the search string cannot cross line boundaries; that is, all the text necessary to match the pattern must be on a single line.

When you use the -r option (on by default), the search string is treated as a regular expression (not a literal expression).

The following characters have special meanings:

Symbol	Description
^	A circumflex at the start of the expression matches the start of a line.
$	A dollar sign at the end of the expression matches the end of a line.
.	A period matches any character.
*	An asterisk after a character matches any number of occurrences of that character followed by any characters, including zero characters. For example, bo* matches bot, boo, as well as bo.
+	A plus sign after a character matches any number of occurrences of that character followed by any characters, except zero characters. For example, bo+ matches bot and boo, but not b, bo, or bt.
{}	Characters or expressions in braces are grouped so that the evaluation of a search pattern can be controlled and so grouped text can be referred to by number.
[]	Characters in brackets match any one character that appears in the brackets, but no others. For example [bot] matches b, o, or t.
[^]	A circumflex at the start of the string in brackets means NOT. Hence, [^bot] matches any characters except b, o, or t.
[-]	A hyphen within the brackets signifies a range of characters. For example, [b-o] matches any character from b through o..
\	A backslash before a wildcard character tells GREP to treat that character literally, not as a wildcard. For example, \^ matches ^ and does not look for the start of a line.

Four of the "special" characters ($, ., *, and +) do not have any special meaning when used within a bracketed set. In addition, the character ^ is only treated specially if it immediately follows the beginning of the set definition (immediately after the [delimiter).

GREP Examples

Example 1

```
grep -r "[^a-z]main\ *\(" *.c
```

Matches:

```
main(i,j:integer)
if (main ()) halt;
if (MAIN ()) halt;
```

Does Not Match:

```
mymain()
```

Explanation: The search string tells GREP to search for the word main with no preceding lowercase letters ([^a-z]), followed by zero or more occurrences of blank spaces (\ *), then a left parenthesis. Since spaces and tabs are normally considered command-line delimiters, you must quote them if you want to include them as part of a regular expression.

Example 2

```
grep -ri [a-c]:\\data\.fil *.c *.inc
```

Matches:

```
A:\data.fil
B:\DATA.FIL
c:\Data.Fil
```

Does Not Match:

```
d:\data.fil a:data.fil
```

Explanation: Because the backslash (\) and period (.) characters usually have special meaning in path and file names, you must place the backslash escape character immediately in front of them if you want to search for them. The -i option is used here, so the search is not case sensitive.

Example 3

```
grep "search string with spaces" *.doc *.c
```

Matches:

```
A search string with spaces in it.
```

Does not match

```
This search string has spaces in it.
```

Explanation: This is an example of how to search for a string containing specific text.

Example 4

```
grep -rd "[ ,.:?'\"]"$ \*.doc
```

Matches:

```
He said hi to me.
Where are you going?
In anticipation of a unique situation,
Examples include the following:
"Many men smoke, but fu man chu."Explanation
```

Does not match:

```
He said "Hi" to me
Where are you going? I'm headed to the
```

Explanation: This example searches for any one of the characters " . : ? ' and , at the end of a line. The double quote within the range is preceded by an escape character so it is treated as a normal character instead of as the ending quote for the string. Also, the $ character appears outside of the quoted string. This demonstrates how regular expressions can be concatenated to form a longer expression.

Example 5

```
grep -w[=] = *.c
```

Matches:

```
i = 5;
j=5;
i += j;
```

Does not match:

```
if (i == t) j++;
/* ================================== */
```

This example redefines the current set of legal characters for a word as the assignment operator (=) only, then does a word search. It matches C assignment statements, which use a single equal sign (=), but not equality tests, which use a double equal sign (==).

See Also

BCC32.EXE (see page 159)

3.1.1.8 ILINK32.EXE, the Incremental Linker

ILINK32 links object modules (.OBJ files), library modules (.LIB files), and resources to produce executable files (.EXE, .DLL, and .BPL files). ILINK32 creates and maintains a series of state files that contains this information. These state files allow subsequent links to be incremental, greatly reducing the total link time.

Command Line Syntax

```
ILINK32 [@<respfile>][<options>] <startup> <myobjs>, [<exe>], [<mapfile>], [<libraries>], [<deffile>], [<resfile>]
```

To display command line help, enter:

```
ilink32
```

Linker command-line options are case-sensitive.

The linker can also use a configuration file called ILINK32.CFG for options that you'd typically type at the command-line.

Element	Description
[@<respfile>]	A response file is an ASCII file that lists linker options and file names that you would normally type at the command line. By placing options and file names in a response file, you can save the amount of keystrokes you need to type to link your application.
<options>	Linker options that control how the linker works. For example, options specify whether to produce an .EXE, .BPL, or .DLL file. Linker options must be preceded by either a slash (/) or a hyphen (-).
<startup>	A CodeGear initialization module for executables or DLLs that arranges the order of the various segments of the program. Failure to link in the correct initialization module usually results in a long list of error messages telling you that certain identifiers are unresolved, or that no stack has been created.
<myobjs>	The .OBJ files you want linked. Specify the path if the files aren't in the current directory. (The linker appends an .OBJ extensions if no extension is present.)
<exe>	The name you want given to the executable file (.EXE, .BPL, or .DLL). If you don't specify an executable file name, ILINK32 derives the name of the executable by appending .EXE, .BPL, or .DLL to the first object file name listed. (The linker assumes or appends an .EXE extensions for executable files if no extension is present. It also appends a .DLL extension for dynamic link libraries if no extension is present. Similarly, the linker appends a .BPL extension for package files if no extension is present.)
<mapfile>	The name you want given to the map file. If you don't specify a name, the map file name is the same as the .EXE file (but with the .MAP extension). (The linker appends a .MAP extension if you do not specify an extension.)
<libraries>	The library files you want included at link time. Do not use commas to separate the libraries listed. If a file is not in the current directory or the search path (see the /L option) then you must include the path in the link statement. (The linker appends a .LIB extension if no extension is present.) The order in which you list the libraries is very important; be sure to use the order defined in this list: 1. Code Guard libraries (if needed) 2. Any of your own user libraries (note that if a function is defined more than once, the linker uses the first definition encountered) 3. IMPORT32.LIB (if you are creating an executable that uses the Windows API) 4. Math libraries 5. Runtime libraries
<deffile>	The module definition file for a Windows executable. If you don't specify a module definition (.DEF) file and you have used the `/Twd` or `/Twe` option, the linker creates an application based on default settings. (The linker appends a .DEF extension if no extension is present.)
<resfile>	A list of .RES files (compiled resource files) to bind to the executable. (The linker appends an .RES extension if no extension is present.)

Linker Options

Option	Description
@xxxx	Uses the response file `xxxx`.
/A:dd	Specifies file alignment (backward compatibility switch, use /Af).
/Af:nnnn	Specifies file alignment; set nnnn in hex or decimal. nnnn must be a power of 2. Default is 0x200 = 512 byte boundaries.
/Ao:nnnn	Specifies object alignment; set nnnn in hex or decimal. nnnn must be a power of 2. Default is 0x1000 = 4096 byte boundaries.
/aa	Builds a 32-bit Windows application.
/ad	Builds a 32-bit Windows device driver.
/ap	Builds a 32-bit Windows console application.
/B:xxxx	Specifies the image base address and removes the relocation table.
/b:xxxx	Specifies the image base address, and preserves DLL relocation table. Successive objects are aligned on 64K linear address boundaries. Improves load-time and runtime performance.Recommended for producing DLLs. Default is Off.
/C	Refreshes the linker state files. Erases all linker state files before each link; then recreates the files and continues with the link.
/c	Treats case as significant in public and external symbols. Default is On.
/d	Delay-loads a DLL. The specified DLL is not loaded or initialized until an entrypoint in the DLL is actually called.
/D:xxxx	Saves the specified description in the PE image.
/Enn	Specifies the max. number of errors.
/GC	Inserts a comment string ino the image directly after the object table in the PE file header. You can specify more than one string. Default is Off.
/GD	Generates a Delphi compatible RC file (DRC file). Default is Off.
/GF:xxxx	Sets image flags. The xxxx flags are: • `-GF SWAPNET` copies the image to a local swap file and runts it from there if the image resides on a network drive. • `-GF SWAPCD` copies the image to a local swap file and runs it if the image resides on removable media (CD, zip drive). • `-GF UNIPROCESSOR` prevents the application from running on a multiprocessor system. • `-GF LARGEADDRESSAWARE` tells the OS that the application understands addresses larger than 4G. • `-GF AGGRESSIVE` aggressively trims the workingg seet of an application when the application is idel (suited for screen savers and other processes that you want to stay out of the way of main line processes).
/Gk	Keeps output files on error. Default is Off.
/Gl	Generates a .LIB file.
/Gpd	Generates design-time-only package. Default is: the package works at both design time and runtime.
/Gpr	Generate runtime-only package Default is: the package works at both design time and runtime.
/Gn	Don't generate state files (disable incremental linking). Subsequent links will take just as long as the first one. Default is Off.

/Gs:string=[EICRWSDKP]	Set section flags. Adds the specified flag(s) to the existing flags for a given section, but cannot remove default flags from a section.
	Section flags are:
	`E = Executable`
	`C = Contains code`
	`I = Contains initialized data`
	`R = Section is readable`
	`W = Section is writable`
	`S = Section is shared`
	`D = Section is discardable`
	`K = Section must not be cached`
	`P = Section must not be paged`
	Example -GS:.rsrc=W This switch makes the RSRC section writable.

/Gz	Calculate checksum of target and insert result in PE header. Used for NT Kernel mode drivers and system DLLs. Default is Off.
/H:xxxx	Specifies application heap reserve size in hex or decimal. Minimum value is 0. This switch overrides the HEAPSIZE sestting in a module definition file. Default is 1 MB (0x1000000).
/HC:nnnn	Specifies application heap commit size.
/I	Specifies the directory for intermediate output files. Directs linker state files, but the MAP file and TDS files are saved in the same directory as the output image, unless otherwise specified for the MAP file.
/j	Specifies object search paths. **Example** ILINK32 /jc:\myobjs;.\objs splash .\common\logo,,,utils logolib This command directs the linker to first search the current directory for SPLASH.OBJ. If the file is not found in the current directory, the linker then searches the C:\MYOBJS directory, and then the .\OBJs directory. However, the linker does not use the object search paths to find the file LOGO.OBJ because an explicit path was given for this file.
/L	Specifies the library search path to use if no explicit path is given for the LIB file and the linker cannot find the LIB file in the current directory. **Example** ILINK32 /Lc:\mylibs;.\libs splash .\common\logo,,,utils logolib directs the linker to first search the current directory for SPLASH.LIB. If the file is not found in the current directory, the linker then searches the C:\MYLIBS directory, and then the .\LIBs directory. However, the linker does not use the library search paths to find the file LOGO.LIB because an explicit path was given for this file.
/M	Prints the mangled C++ identifiers in the map file, not the full name.
/m	The segments listing has a line for each segment, showing the segment starting address, segment length, segment name, and the segment class. The public symbols are broken down into two lists, the first showing the symbols in sorted alphabetically, the second showing the symbols in increasing address order. Symbols with absolute addresses are tagged Abs. A list of public symbols is useful when debugging.:Many debuggers use public symbols, which lets you refer to symbolic addresses while debugging.
/q	Suppresses command line banner.
/r	Verbose linking. Default is Off.
/Rr	Replaces resources.
/S:xxxx	Specifies application stack reserve size.
/Sc:xxxx	Specifies application stack commit size.
/s	Produces a detailed map file of segments.
/Tpd	Targets a 32-bit Windows DLL file.

/Tpe	Targets a 32-bit Windows EXE file.
/Tpp	Generates a package.
/t	Displays time spent on link.
/w	Turns all warnings on.
/w-dee	Disables the warning: ".EXE module built with a .DLL or BPL extension."
/w-dpl	Disables the warning: "Duplicate symbol in library.".
/w-dup	Disables the warning: "Duplicate symbol."
/w-exp	Enables the warning: "Attempt to export non-public symbol.."
/w-nou	Disables the warning: "Creating a package without any units."
/w-rty	Enables the warning: "Unable to perform incremental link - performing full link".
/w-srd	Disables the warning: "Stripping relocations from a DLL may cause it to malfunction."
/w-snf	Disables the warning: "Section not found."
/w-uld	Disables the warning: "Unable to load DLL."
/x	Suppresses creation of a MAP file.
(blank)	Default map file of segments (no linker switch, map file created by default)

Linker Response Files

You can use response files with the command-line linker to specify linker options.

Response files are ASCII files that list linker options and file names that you would normally type at the command line. Response files allow you longer command lines than most operating systems support, plus you don't have to continually type the same information. Response files can include the same information as configuration files, but they also support the inclusion of file names.

Unlike the command line, a response file can be several lines long. To specify an added line, end a line with a plus character (+) and continue the command on the next line. Note that if a line ends with an option that uses the plus to turn it on (such as /v+), the + is not treated as a line continuation character (to continue the line, use /v+ +).

If you separate command-line components (such as .OBJ files from .LIB files) by lines in a response file, you must leave out the comma used to separate them on the command line.

For example:

```
/c c0ws+
myprog,myexe +
mymap +
mylib cws
```

leaves out the commas you'd have to type if you put the information on the command line:

```
ILINK32 /c c0ws myprog,myexe,mymap,mylib cws
```

To use response files:

1. Type the command-line options and file names into an ASCII text file and save the file. Response files typically have an .RSP extension.
2. Type: ILINK32 @[<path>]<RESFILE.RSP> where <RESFILE.RSP> is the name of your response file.

You can specify more than one responses file, as follows:

```
ilink32 /c @listobjs.rsp,myexe,mymap,@listlibs.rsp
```

Note: You can add comments to response files using semicolons; the linker ignores any text on a line that follows a semicolon.

Module Definition Files

The module definition file is an ASCII text file that provides information to ILINK32 about the contents and system requirements of a Windows application. You can create a module definition file using IMPDEF.EXE, and you can create import libraries from module definition files using IMPLIB.EXE.

If no module definition file is specified, the following defaults are assumed:

```
CODE        PRELOAD MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE MULTIPLE    (for applications)
            PRELOAD MOVEABLE SINGLE    (for DLLs)
HEAPSIZE    4096
STACKSIZE   1048576
```

To change an application's attributes from these defaults, you need to create a module definition file.

If you delete the EXETYPE statement, the linker can determine what kind of executable you want to produce from the options you supply on the command line.

You can include an import library to substitute for the IMPORTS section of the module definition.

You can use the __declspec(dllexport) or _export keywords in the definitions of export functions in your C and C++ source code to remove the need for an EXPORTS section. Note, however, that if __declspec(dllexport) or _export is used to export a function, that function is exported by name rather than by ordinal. Please also note that __declspec(dllexport) is the preferred method of export

Linker-State Files

The four linker-state files have file names of <Project>.IL?, where <Project> is taken from the name of the project .BPR file, and the character in place of the question mark (?) identifies each of the four individual linker-state files.

Debug Information File

If you include debug information in your final linked executable, ILINK32 will always store the debug information in a separate .TDS file, named <Project Name>.TDS by default. The debugger should be able to read this debugger-information file. ILINK32 will always create this file. If you do not have the -v linker switch set, the TDS file is marked as invalid.

See Also

RLINK32.EXE (see page 204)

IMPDEF.EXE (see page 179)

Module Definition Files (see page 199)

3.1.1.9 IMPDEF.EXE, the Module Definition Manager

Import libraries provide access to the functions in a Windows DLL. Import libraries contain records. Each record contains the name of a DLL and specifies where in the DLL the imported functions reside. These records are bound to the application by the linker and provide Windows with the information necessary to resolve DLL function calls. You can substitute an import library for part or all of the IMPORTS section of a module definition file.

IMPDEF takes as input a DLL name, and produces as output a module definition file with an EXPORTS section containing the names of functions exported by the DLL.

Command Line Syntax

```
IMPDEF <options> <destination.def> <source.dll>
```

Option	Description
-a	Adds '_' alias for cdecl functions for compatibility with Microsoft libraries
-h	Emits hints

For example:

```
IMPDEF DestName.DEF SourceName.DLL
```

To display command line help, enter:

```
impdef
```

This creates a module definition file named DestName.DEF from the file SourceName.DLL. The resulting module definition file would look something like this:

```
LIBRARY     <FileName>
DESCRIPTION '<Description>'
EXPORTS
            <ExportFuncName>              @<Ordinal>
        .
        .
        .
            <ExportFuncName>              @<Ordinal>
```

- **<FileName>** is the root file name of the DLL.
- **'<Description>'** is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement.
- **<ExportFuncName>** names an exported function.
- **<Ordinal>** is that function's ordinal value (an integer).

Classes in a DLL

IMPDEF is useful for a DLL that uses C++ classes. If you use the __declspec (or _export) keyword when defining a class, all of the non-inline member functions and static data members for that class are exported. It's easier to let IMPDEF make a module definition file for you because it lists all the exported functions, and automatically includes the member functions and static data members.

Since the names of these functions are mangled, it would be tedious to list them all in the EXPORTS section of a module definition file simply to create an import library from the module definition file. If you use IMPDEF to create the module definition file, it will include the ordinal value for each exported function. If the exported name is mangled, IMPDEF will also include that function's unmangled, original name as a comment following the function entry. So, for instance:

```
LIBRARY     <FileName>
DESCRIPTION '<Description>'
EXPORTS
            <MangledExportFuncName>  @<Ordinal> ; <ExportFuncName>
        .
        .
        .
            <MangledExportFuncName>  @<Ordinal> ; <ExportFuncName>
```

- **<FileName>** is the root file name of the DLL.
- **'<Description>'** is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement.
- **<MangledExportFuncName>** provides the mangled name.
- **<Ordinal>** is that function's ordinal value (an integer).
- **<ExportFuncName>** gives the function's original name.

Functions in a DLL

IMPDEF creates an editable source file that lists all the exported functions in the DLL. You can edit this .DEF file to contain only those functions that you want to make available to a particular application, then run IMPLIB on the edited .DEF file. This results in an import library that contains import information for a specific subset of a DLL's export functions.

Suppose you're distributing a DLL that provides functions to be used by several applications. Every export function in the DLL is defined with __declspec (or _export). Now, if all the applications used all the DLL's exports, then you could use IMPLIB to make one import library for the DLL. You could deliver that import library with the DLL, and it would provide import information for all of the DLL's exports. The import library could be linked to any application, thus eliminating the need for the particular application to list every DLL function it uses in the IMPORTS section of its module definition file.

But suppose you want to give only a few of the DLL's exports to a particular application. Ideally, you want a customized import library to be linked to that application--an import library that provides import information only for the subset of functions that the application will use. All of the other export functions in the DLL will be hidden to that client application.

To create an import library that satisfies these conditions, run IMPDEF on the compiled and linked DLL. IMPDEF produces a module definition file that contains an EXPORT section listing all of the DLL's export functions. You can edit that module definition file, remove the EXPORTS section entries for those functions you don't want in the customized import library, and then run IMPLIB on the module definition file. The result will be an import library that contains import information for only those export functions listed in the EXPORTS section of the module definition file.

See Also

ILINK32.EXE (see page 174)

IMPLIB.EXE (see page 181)

Module Definition Files (see page 199)

3.1.1.10 IMPLIB.EXE, the Import Library Tool

IMPLIB takes as input either DLLs or module definition files, or both, and produces an import library as output.

If you've created a Windows application, you've already used IMPORT32.LIB, the import library for the standard Windows DLLs. IMPORT32.LIB is linked automatically when you build a Win32 application in the C++Builder IDE and when using BCC32 at the command line.

An import library lists some or all of the exported functions for one or more DLLs. IMPLIB creates an import library directly from DLLs or from module definition files for DLLs (or a combination of the two).

Command Line Syntax

```
IMPLIB <options> <LibName> [< DefFiles>... | <DLLs>... ] [@<ResponseFile> | <sourcename> ]
[<sourcename> ...]
```

To display command line help, enter:

```
implib
```

Command Line Element	Description
<options>	An optional list of one or more of the following IMPLIB command options: • **-a** Adds '_' alias for cdecl functions for compatibility with Microsoft libraries. If the identifier already begins with an underscore (_), it is skipped (unless you use the -aa option). • **-aa** Forces the creation of an '_' alias for cdecl functions for compatibility with Microsoft libraries. If the identifier begins with an underscore (_), the -aa option adds a second underscore. • **-c** Emits warning on case sensitive symbols. • **-f** Forces imports by name. • **-w** Emits no warnings.
<LibName>	The name for the new import library.
<DefFiles>	One or more existing module definition files for one or more DLLs. You must specify at least one DLL or module definition file.
<DLLs>	One or more existing DLLs. You must specify at least one DLL or one module definition file.
@<ResponseFile>	An ACSII text file that contains a list of DEF and DLL files that you want to process using IMPLIB. In the response file, separate the filenames using either spaces or new lines.
<sourcename>	Either a DEF or DLL file that is to be processed by IMPLIB.

Example

```
implib foo.lib @respon.txt
```

Note: A DLL can also have an extension of .EXE or .DRV, not just .DLL.

See Also

IMPDEF.EXE (see page 179)

3.1.1.11 Using Include Files

In C++, include files always have the file extension .h.

Include File Search Algorithms

The CodeGear C++ compiler searches for files included in your source code with the #include directive in the following ways:

- If you specify a path and/or directory with your include statement, the compiler searches only the specified location. For example, if you have the following statement in your code: `#include "c:\C++\include\vcl\vcl.h"` the header file vcl.h must reside in the directory C:\C++\include. In addition, if you use the statement: `#include <vcl\vcl.h>` and you set the Include option (-I) to specify the path C:\C++\include, the file vcl.h must reside in C:\C++\include\vcl, and not in C:\C++include or C:\vcl.
- If you put a #include <somefile> statement in your source code, the compiler searches for "somefile" only in the directories specified with the Include (-I) option.
- If you put a #include "somefile" statement in your code, the compiler will search for "somefile" in the following order:

1. The same directory as the file containing the #include statement.
2. The directories of flies that include (#include) that file.
3. The current directory
4. The directories specified with the include (-I) option

Library File Search Algorithms

The library file search algorithms are similar to those for include files:

- **Implicit libraries:** The CodeGear C++ compiler searches for implicit libraries only in the specified library directories; this is similar to the search algorithm for: `#include "somefile"` Implicit library files are the ones the compiler automatically links in and the start-up object file (C0x.OBJ).
- **Explicit libraries:** Where the compiler searches for explicit (user-specified) libraries depends in part on how you list the library file name. Explicit library files are ones you list on the command line or in a project file; these are file names with a .LIB extension.
- If you list an explicit library file name with no drive or directory (such as MYLIB.LIB), the compiler first searches for that library in the current directory. If the first search is unsuccessful, the compiler looks in the directories specified with the Library (-L) option. This is similar to the search algorithm for #include "somefile".
- If you list a user-specified library with drive and/or directory information (like this: c:\mystuff\mylib1.lib), the CodeGear C++ compiler searches only in the location you explicitly listed as part of the library path name and not in any specified library directories.

See Also

BCC32.EXE (see page 159)

3.1.1.12 MAKE

MAKE.EXE is a command-line utility that helps you manage project compilation and link cycles. MAKE is not inherently tied to compiling and linking, but is a more generic tool for executing commands based on file dependencies. MAKE helps you quickly build projects by compiling only the files you have modified since the last compilation. In addition, you can set up rules that specify how MAKE should deal with the special circumstances in your builds.

MAKE Basics

MAKE uses rules you write along with its default settings to determine how it should compile the files in your project. For example, you can specify when to build your projects with debug information and to compile your .OBJ files only if the date/time stamps of a source file is more recent than the .OBJ itself. If you need to force the compilation of a module, use TOUCH.EXE to modify the time stamp of one of the module's dependents.

In an ASCII makefile, you write explicit and implicit rules to tell MAKE how to treat the files in your project; MAKE determines if it should execute a command on a file or set of files using the rules you set up. Although your commands usually tell MAKE to compile or link a set of files, you can specify nearly any operating system command with MAKE.

Command Line Syntax

```
MAKE [<options>...][<target>[<target>]]
```

You must separate the MAKE command and the options and target arguments with spaces.

When specifying targets, you can use wildcard characters (such as * and ?) to indicate multiple files.

To display command line help, enter:

```
make –?
```

or:

```
make –h
```

Command Line Element	Description
[<options>]	MAKE options that control how MAKE works. See "Make Command Options" in this topic.

<target>	The name of the target listed in the makefile that you want to build

Default MAKE Actions

When you issue a MAKE command, MAKE looks for the file BUILTINS.MAK, a file that you create to contain the default rules for MAKE (use the -r option to ignore the default rules). MAKE looks for this file first in the current directory, then in the directory where MAKE.EXE is stored. After loading BUILTINS.MAK, MAKE looks in the current directory for a file called MAKEFILE or MAKEFILE.MAK (use the -f option to specify a file other than MAKEFILE). If MAKE can't find the makefile, it generates an error message.

After loading the makefile, MAKE tries to build only the first explicit target listed in the makefile by checking the time and date of the dependent files of the first target. If the dependent files are more recent than the target file, MAKE executes the commands to update the target.

If one of the first target's dependent files is used as a target elsewhere in the makefile, MAKE checks that target's dependencies and builds it before building the first target. This chain reaction is called a linked dependency.

If something during the build process fails, MAKE deletes the target file it was building. Use the `precious` directive if you want MAKE to keep the target when a build fails.

You can stop MAKE after issuing the MAKE command by pressing Ctrl+Break or Ctrl+C

About makefiles

A makefile is an ASCII file that contains the set of instructions that MAKE uses to build a certain project. Although MAKE assumes your makefile is called MAKEFILE or MAKEFILE.MAK, you can specify a different makefile name with the -f option.

MAKE either builds the target(s) you specify with the make command or it builds the first target it finds in the makefile. To build more than a single target, use a symbolic target in your makefile.

Makefiles can contain:

- Comments (precede a comment with a number sign [#])
- Explicit and implicit rules
- Macros
- Directives

Symbolic Targets

A symbolic target forces MAKE to build multiple targets in a makefile. When you specify a symbolic target, the dependency line lists all the targets you want to build (a symbolic target basically uses linked dependencies to build more than one target).

For example, the following makefile uses the symbolic target AllFiles to build both FILE1.EXE and FILE2.EXE:

```
AllFiles: file1.exe file2.exe #Note that AllFiles has no commands file1.exe: file1.obj
bcc32 file1.obj file2.exe: file2.obj        bcc32 file2.obj
```

Rules for symbolic targets

- Do not type a line of commands after the symbolic target line.
- A symbolic target must have a unique name; it cannot be the name of a file in your current directory.
- Symbolic target names must follow the operating system rules for naming files.

BUILTINS.MAK

You can create the optional file BUILTINS.MAK and store in it the standard rules and macros that you want MAKE to use when it builds the targets in a makefile. To ignore this file when it exists, use the -r MAKE option.

Here is a sample of a BUILTINS.MAK file:

```
#
# CodeGear C++ BUILTINS.MAK
#

CC = bcc32
RC = brcc32
AS = tasm32

.asm.obj:
    $(AS) $(AFLAGS) $&.asm

.c.exe:
    $(CC) $(CFLAGS) $&.c

.c.obj:
    $(CC) $(CFLAGS) /c $&.c

.cpp.exe:
    $(CC) $(CFLAGS) $&.cpp

.cpp.obj:
    $(CC) $(CPPFLAGS) /c $&.cpp

.rc.res:
    $(RC) $(RFLAGS) /r $&

.SUFFIXES: .exe .obj .asm .c .res .rc

!if !$d(BCEXAMPLEDIR)
BCEXAMPLEDIR = $(MAKEDIR)\..\EXAMPLES

!endif
```

MAKE Command Options

You can use command-line options to control the behavior of MAKE. MAKE options are case-sensitive and must be preceded with either a hyphen (-) or slash (/).

You must separate the MAKE command and the options and target arguments with spaces. When specifying targets, you can use wildcard characters (such as * and ?) to indicate multiple files. To get command-line help for MAKE, type MAKE -?.

For example, to use a file called PROJECTA.MAK as the makefile, type:

```
MAKE -fPROJECTA.MAK
```

Many of the command-line options have equivalent directives that you can use within the makefile.

Option	Description
-a	Checks dependencies of include files and nested include files associated with .OBJ files and updates the .OBJ if the .h file changed. See also -c.
-B	Builds all targets regardless of file dates.

-c	Caches autodependency information, which can improve MAKE speed. Use with -a. Do not use this option if MAKE modifies include files (which can happen if you use TOUCH in the makefile or if you create header or include files during the MAKE process).
-D<macro>	Defines macro as a single character, causing an expression !ifdef macro written in the makefile to return true.
-D<macro>=<string>	Defines <macro> as <string>. If <string> contains any spaces or tabs, enclose <string> in quotation marks. The -D is optional.
-d<directory>	Specifies the drive and the directory that MAKER (the real mode version of MAKE) uses when it swaps out of memory. This option must be used with -S. MAKE ignores this option.
-e	Ignores a macro if its name is the same as an environment variable. MAKE uses the environment variable instead of the macro.
-f<filename>	Uses <filename> or <fillename>.MAK instead of MAKEFILE (a space after -f is optional).
-h or: -?	Help. Displays the MAKE options. Default settings are shown with a trailing plus sign.
-I<directory>	Searches for include files in the current directory first, then in the specified directory.
-i	Ignores the exit status of all programs run from the makefile and continues the build process.
-K	Keeps temporary files that MAKE creates (MAKE typically deletes temporary files).
-m	Displays the date and timestamp of each file as MAKE processes it.
-n	Prints the MAKE commands but does not perform them. This is helpful for debugging makefiles.
-N	Causes MAKE to mimic Microsoft's NMAKE.
-p	Displays all macro definitions and implicit rules before executing the makefile.
-q	Returns 0 if the target is up-to-date and nonzero if it is not up-to-date (for use with batch files).
-r	Ignores any rules defined in BUILTINS.MAK.
-s	Suppresses onscreen command display (silent).
-S	Swaps MAKER out of memory while commands are executed, reducing memory overhead and allowing compilation of large modules. MAKE ignores this option.
-U<macro>	Undefines the previous macro definition of <macro>.
-W<filename>	Writes MAKE to <filename>, updating all non-string options.

See Also

MAKE Macros (see page 191)

MAKE Directives (see page 186)

MAKE Rules and Commands (see page 193)

3.1.1.13 MAKE Directives

MAKE directives resemble directives in languages such as C and Pascal. In MAKE, directives perform various control functions, such as displaying commands onscreen before executing them. MAKE directives begin either with an exclamation point or a period, and they override any options given on the command line. Directives that begin with an exclamation point must appear at the start of a new line.

The following table lists the MAKE directives and their corresponding command-line options:

Directive	Option (if available)	Description/Example
.autodepend	-a	Turns on autodependency checking. Autodependencies are the files that are automatically included in the targets you build, such as the header files included in your C++ source code. With .autodepend on, MAKE compares the dates and times of all the files used to build the .OBJ, including the autodependency files. If the dates or times of the files used to build the .OBJ are newer than the date/time stamp of the .OBJ file, the .OBJ file is recompiled. You can use .autodepend (or -a) in place of forming linked dependencies.
.cacheautodepend	-c	Turns on autodependency caching.
!cmdswitches		Uses + or - followed by non-string option letters to turn each option on or off. Spaces or tabs must appear before the + or - operator, none can appear between the operator and the option letters.
!elif		Acts like a C else if.
!else		Acts like a C else.
!endif		Ends an !if, !ifdef, or !ifndef statement

!error		Stops MAKE and prints an error message The syntax of the !error directive is: `!error <message>`
		MAKE stops processing and prints the following string when it encounters this directive:
		`Fatal makefile exit code: Error directive: <message>`
		Embed !error in conditional statements to abort processing and print an error message, as shown in the following example: `!if !$d(MYMACRO) #if MYMACRO isn't defined !error MYMACRO isn't defined ! endif`
		If MYMACRO isn't defined, MAKE terminates and prints:
		`Fatal makefile 4: Error directive: MYMACRO isn't defined`
		Error-Checking Controls: MAKE offers four different error-checking controls:
		• The .ignore directive turns off error checking for a selected portion of the makefile.
		• The -i command-line option turns off error checking for the entire makefile.
		• The -num prefix, which is entered as part of a rule, turns off error checking for the related command if the exit code exceeds the specified number.
		• The - prefix turns off error checking for the related command regardless of the exit code.

!if		Begins a conditional statement.
!ifdef		Acts like a C #ifdef, testing whether a given macro has been defined.
!ifndef		Acts like a C #ifndef, testing whether a given macro is undefined.
.ignore	-i	MAKE ignores the return value of a command.
!include		Acts like a C #include, specifying a file to include in the makefile. This directive is like the #include preprocessor directive for the C or C++ language -- it lets you include the text of another file in the makefile: !include <filename> You can enclose the filename in quotation marks (" ") or angle brackets (< >) and nest directives to unlimited depth, but writing duplicate !include directives in a makefile isn't permitted -- you'll get the error message cycle in the include file. Rules, commands, or directives must be complete within a single source file; you can't start a command in an !include file, then finish it in the makefile. MAKE searches for !include files in the current directory unless you've specified another directory with the -I command-line option.

.keep	-K	Keeps temporary files that MAKE creates (MAKE usually deletes them).

!message		Prints a message to stdout while MAKE runs the makefile
		The !message directive lets you send messages to the screen from a makefile. You can use these messages to help debug a makefile that isn't working properly. For example, if you're having trouble with a macro definition, you could put this line in your makefile:
		`!message The macro is defined here as: $(MacroName)`
		When MAKE interprets this line, it will print onscreen (assuming the macro expands to .CPP):
		`The macro is defined here as: .CPP`

.noautodepend	**a**	Turns off autodependency checking.
.nocacheautodepend	**c**	Turns off autodependency caching.
.noIgnore	**i**	Turns off .Ignore.
.nokeep	**K**	Does not keep temporary files that MAKE creates.
.nosilent	**s**	Displays commands before MAKE executes them.
.noswap	**S**	Tells MAKE not to swap iteself out of memory before executing a command.
.path.ext		Tells MAKE to search for files with the extension .ext in path directories To tell MAKE to look for files with the .c extension in C:\SOURCE or C:\CFILES and to look for files with the .obj extension in C:\OBJS: `.path.c = C:\CSOURCE;C:\CFILES .path.obj = C:\OBJS`
.precious		Saves the target or targets even if the build fails If a MAKE build fails, MAKE deletes the target file. The .precious directive prevents the file deletion, which you might desire for certain kinds of targets. For example, if your build fails to add a module to a library, you might not want the library to be deleted. The syntax for .precious is: `.precious: <target> [<target>...]`
.silent	-s	MAKE executes commands without printing them first.

.suffixes		Determines the implicit rule for ambiguous dependencies. The .suffixes directive tells MAKE the order (by file extensions) for building implicit rules. The syntax of .suffixes is:
		`.suffixes: .<ext> [.<ext> ...]`
		where .<ext> represents the dependent file extensions you list in your implicit rules.
		For example, you could include the line `.suffixes: .asm .c .cpp` to tell MAKE to interpret implicit rules beginning with the ones dependent on .ASM files, then .C files, then .CPP files, regardless of what order they appear in the makefile.
		The following .suffixes example tells MAKE to look for a source file first with an .ASM extension, next with a .C extension, and finally with a .CPP extension. If MAKE finds MYPROG.ASM, it builds MYPROG.OBJ from the assembler file by calling TASM. MAKE then calls ILINK32; otherwise, MAKE searches for MYPROG.C to build the .OBJ file or it searches for MYPROG.CPP.
		`.suffixes: .asm .c .cpp`
		`myprog.exe: myprog.obj`
		`bcc32 myprog.obj`
		`.cpp.obj:`
		`bcc32 -P -c $<`
		`.asm.obj:`
		`tasm /mx $`
		`.c.obj:`
		`bcc32 -P- -c $<`

.swap	-S	Tells MAKE to swap itself out of memory before executing a command.
!undef		Clears the definition of a macro. After this, the macro is undefined. !undef (undefine) causes an !ifdef MacroName test to fail. The syntax of the !undef directive is: `!undef MacroName`

Using Macros in Directives

You can use the $d macro with the !if conditional directive to perform some processing if a specific macro is defined. Follow the $d with macro name enclosed in parentheses or braces, as shown in the following example:

```
!if $d(DEBUG)           #If DEBUG is defined,
bcc32 -v f1.cpp f2.cpp   #compile with debug information;
!else                   #otherwise
bcc32 -v- f1.cpp f2.cpp   #don't include debug information.
!endif
```

Null Macros

you can substitute the characters .obj for the characters .cpp by using the following MAKE command:

```
NULLMACRO =
```

Either of the following lines can define a null macro on the MAKE command line:.

```
NULLMACRO =""
-DNULLMACRO
```

!if and Other Conditional Directives

The !if directive works like C if statements. As shown here, the syntax of !if and the other conditional directives resembles compiler conditionals:

!if condition	!if condition	!if condition	!ifdef macro
!endif	!else	!elif condition	!endif
	!endif	!endif	

The following expressions are equivalent:

```
!ifdef macro and !if $d(macro)
ifndef macro and !if !$d(macro)
```

These rules apply to conditional directives:

- One !else directive is allowed between !if, !ifdef, or !ifndef and !endif directives.
- Multiple !elif directives are allowed between !if, !ifdef, or !ifndef, !else and !endif.
- You can't split rules across conditional directives.
- You can nest conditional directives.
- !if, !ifdef, and !ifndef must have matching !endif directives within the same file.

The following information can be included between !if and !endif directives:

- Macro definition
- Explicit rule
- Implicit rule
- Include directive
- !error directive
- !undef directive

In an if statement, a conditional expression consists of decimal, octal, or hexadecimal constants and the operators shown in the following table:

Operator	Description	Operator	Description
-	Negation	?:	Conditional expression
~	Bit complement	!	Logical NOT
+	Addition	>>	Right shfit
-	Subtraction	<<	Left shift
*	Multiplication	&	Bitwise AND
/	Division	|	Bitwise OR
%	Remainder	^	Bitwise XOR
&&	Logical AND	>+	Greater than or equal to *
||	Logical OR	<+	Less than or equal to *
>	Greater than	==	Equality *
<	Less than	!=	Inequality *

- The Operator also works with string expressions.

MAKE evaluates a conditional expression as either a 32-bit signed integer or a character string.

See Also

Using MAKE.EXE (see page 183)

MAKE Macros (see page 191)

MAKE Rules and Commands (see page 193)

3.1.1.14 MAKE Macros

A macro is a variable that MAKE expands into a string whenever MAKE encounters the macro in a makefile. For example, you can define a macro called LIBNAME that represents the string "mylib.lib." To do this, type the line LIBNAME = mylib.lib at the beginning of your makefile. Then, when MAKE encounters the macro $(LIBNAME), it substitutes the string mylib.lib. Macros let you create template makefiles that you can change to suit different projects.

To use a macro in a makefile, type $(MacroName) where MacroName is a defined macro. You can use either braces or parentheses to enclose MacroName.

MAKE expands macros at various times depending on where they appear in the makefile:

- Nested macros are expanded when the outer macro is invoked.
- Macros in rules and directives are expanded when MAKE first looks at the makefile.
- Macros in commands are expanded when the command is executed.

If MAKE finds an undefined macro in a makefile, it looks for an operating system environment variable of that name (usually defined with SET) and uses its definition as the expansion text. For example, if you wrote $(PATH) in a makefile and never defined PATH, MAKE would use the text you defined for PATH in your AUTOEXEC.BAT. See your operating system manuals for information on defining environment variables.

Syntax

```
<MacroName> = <expansion_text>
```

Element	Description
<MacroName>	Is case-sensitive (MACRO1 is different from Macro1). Limited to 512 characters.
<expansion_text>	Is limited to 4096 characters. Expansion characters may be alphanumeric, punctuation, or spaces.

You must define each macro on a separate line in your makefile and each macro definition must start on the first character of the line. For readability, macro definitions are usually put at the top of the makefile. If MAKE finds more than one definition of MacroName, the new definition overwrites the old one.

You can also define macros using the -D command-line option. No spaces are allowed before or after the equal sign (=); however, you can define more than one macro by separating the definitions with spaces. The following examples show macros defined at the command line:

```
make -Dsourcedir=c:\projecta
make -Dcommand="bcc32 -c"
make -Dcommand=bcc32 option=-c
```

Note: Macros defined in makefiles overwrite macros defined on the command line.

String Substitution in MAKE Macros

MAKE lets you temporarily substitute characters in a previously defined macro. For example, if you define the following macro:

```
SOURCE = f1.cpp f2.cpp f3.cpp
```

you can substitute the characters .obj for the characters .cpp by using the following MAKE command:

```
$(SOURCE:.cpp=.obj)
```

This substitution does not redefine the macro.

Rules for macro substitution:

- Syntax: `$(MacroName:original_text=new_text)`

- No space before or after the colon
- Characters in `original_text` must exactly match the characters in the macro definition (text is case-sensitive)

MAKE also lets you use macros within substitution macros. For example:

```
MYEXT=.C
SOURCE=f1.cpp f2.cpp f3.cpp
$(SOURCE:.cpp=$(MYEXT)) #Changes 'f1.cpp' to 'f1.C', etc.
```

The caret (^) symbol causes MAKE to interpret the next character literally. This is useful for inserting a new-line character. For example:

```
MYEXT=.C
SOURCE=f1.cpp f2.cpp f3.cpp
($(SOURCE):.cpp=$(MYEXT)^
)         # changes 'f1.cpp f2.cpp f3.cpp' to:
        # f1.C
        # f2.C
        # f3.C
```

Here, the caret tells MAKE to change each occurrence of .cpp to .C followed by the new-line character.

You can use this to create explicit rules that generate response files for TLIB.EXE. For example:

```
myfile.lib: file1.obj file2.obj file3.obj
TLIB $@ @&&!
+-$(**: = &^
+-)
!
```

Here, MAKE substitutes all space characters in the dependency list with a space followed by an ampersand (&) followed by the new-line character followed by "+-". The resulting response file looks like this:

```
+-file1.obj &
 +-file2.obj &
 +-file3.obj &
 +-
```

Default MAKE Macros

MAKE contains several default macros you can use in your makefiles. The following table lists the macro definition and what it expands to in explicit and implicit rules.

Macro	Expands in Implicit Rule	Expands in Explicit Rule
$*	path\dependent file	path\target file
$<	path\dependent file+ext	path\target file+ext
$:	path for dependents	path for target
$.	dependent file+ext	target file + ext
$&	dependent file	target file
$**	path\dependent file+ext	all dependents file+ext
$?	path\dependent file+ext	old dependents

Macro	Expands to	Comment
_ _MSDOS_ _	1	If running under DOS

_ _MAKE_ _	0x0370	MAKE's hex version number
MAKE	make	MAKE's executable file name
MAKEFLAGS	options	The options typed on the command line
MAKEDIR	directory	Directory where MAKE.EXE is located

Modifying Default MAKE Macros

If the default macros don't give you the exact string you want, macro modifiers let you extract parts of the string to suit your purpose. Macro modifiers are usually used with $< or $@.

To modify a default macro, use this syntax:

```
$(MacroName [modifier])
```

The following table lists macro modifiers and provides examples of their use:

Modifier	Part of File Name Expanded	Example	Result
D	Drive and directory	$(<D)	C:\PROJECTA\
F	Base and extension	$(<F)	MYSOURCE.C
B	Base only	$(<B)	MYSOURCE
R	Drive, directory, and base	$(<R)	C:\PROJA\SOURCE

See Also

Using MAKE.EXE (see page 183)

MAKE Directives (see page 186)

MAKE Rules and Commands (see page 193)

3.1.1.15 MAKE Rules (Explicit and Implicit) and Commands

You write explicit and implicit rules to instruct MAKE how to build the targets in your makefile. In general, these rules are defined as follows:

- Explicit rules are instructions for specific files.
- Implicit rules are general instructions for files without explicit rules.

All the rules you write follow this general format:

```
Dependency line
  Command line
```

While the dependency line has a different syntax for explicit and implicit rules, the command line syntax stays the same for both rule types.

MAKE supports multiple dependency lines for a single target, and a single target can have multiple command lines. However, only one dependency line should contain a related command line. For example:

```
Target1: dependent1 dep2 dep3 dep4 dep5
Target1: dep6 dep7 dep8
    bcc32 -c $**
```

Explicit Rule Syntax

Explicit rules specify the instructions that MAKE must follow when it builds specific targets. Explicit rules name one or more targets followed by one or two colons. One colon means one rule is written for the target(s); two colons mean that two or more

rules are written for the target(s).

Explicit rules follow this syntax:

```
<target> [<target>...]:[:][{<path>}] [<dependent[s]>...]
    [<commands>]
```

Element	Description
<target>	Specifies the name and extension of the file to be built (a target must begin a line in the makefile -- you cannot precede the target name with spaces or tabs). To specify more than one target, separate the target names with spaces or tabs. Also, you cannot use a target name more than once in the target position of an explicit rule.
<path>	Is a list of directories that tells MAKE where to find the dependent files. Separate multiple directories with semicolons and enclosed the entire path specification in braces.
<dependent>	Is the file (or files) whose date and time MAKE checks to see if it is newer than target. Each dependent file must be preceded by a space. If a dependent appears elsewhere in the makefile as a target, MAKE updates or creates that target before using the dependent in the original target (this in known as a linked dependency).
<commands>	Are any operating system command or commands. You must indent the command line by at least one space or tab, otherwise they are interpreted as a target. Separate multiple commands with spaces.

If a dependency or command line continues on the following line, use a backslash (\) at the end of the first line to indicate that the line continues. For example:

```
MYSOURCE.EXE: FILE1.OBJ\     #Dependency line
FILE3.OBJ               #Dependency line continued
bcc32 file1.obj file3.obj  #Command line
```

Single Targets with Multiple Rules

A single target can have more than one explicit rule. To specify more than a single explicit rule, use a double colon (::) after the target name.

The following example shows targets with multiple rules and commands:

```
.cpp.obj:
   bcc32 -c -ncobj $<

.asm.obj:
   tasm /mx $<, asmobj\

mylib.lib :: f1.obj f2.obj #double colon specifies multiple rules
   echo Adding C files
   tlib mylib -+cobjf1 -+cobjf2

mylib.lib :: f3.obj f4.obj
   echo Adding ASM files
   tlib mylib -+asmobjf3 -+asmobjf4
```

Implicit Rule Syntax

An implicit rule specifies a general rule for how MAKE should build files that end with specific file extensions. Implicit rules start with either a path or a period. Their main components are file extensions separated by periods. The first extension belongs to the dependent, the second to the target.

If implicit dependents are out-of-date with respect to the target, or if the dependents don't exist, MAKE executes the commands associated with the rule. MAKE updates explicit dependents before it updates implicit dependents.

Implicit rules follow this basic syntax:

```
[{>source_dir>}].<source_ext>[{target_dir}].<target_ext>:
    [<commands>]
```

Element	Description
<source_dir>	Specifies the directory (or directories) containing the dependent files. Separate multiple directories with a semicolon.
.<source_ext>	Specifies the dependent filename extension.
<target_dir>	Specifies the directory where MAKE places the target files. The implicit rule will only be used for targets in this directory. Without specifying a target directory, targets from any directory will match the implicit rule.
.<target_ext>	Specifies the target filename extension. Macros are allowed here.
: (colon)	Marks the end of the dependency line.
<commands>	Are any operating system command or commands. You must indent the command line by at least one space or tab; otherwise they are interpreted as a target.

If two implicit rules match a target extension but no dependent exists, MAKE uses the implicit rule whose dependent's extension appears first in the .SUFFIXES list.

Explicit Rules with Implicit Commands

A target in an explicit rule can get its command line from an implicit rule. The following example shows an implicit rule followed by an explicit rule without a command line:

```
.c.obj:
    bcc32 -c $<    #This command uses a macro $< described later
myprog.obj:       #This explicit rule uses the command: bcc32 -c   myprog.c
```

The implicit rule command tells MAKE to compile MYPROG.C (the macro $< replaces the name myprog.obj with myprog.c).

MAKE Command Syntax

Commands immediately follow an explicit or implicit rule and must begin on a new line with a space or tab. Commands can be any operating system command, but they can also include MAKE macros, directives, and special operators that your operating system won't recognize (however, note that | can't be used in commands).

Here are some sample commands:

```
cd..
bcc32 -c mysource.c
COPY *.OBJ C:\PROJECTA
bcc32 -c $(SOURCE) #Macros in "Using MAKE macros"
```

Commands follow this general syntax:

```
[<prefix>...] <commands>
```

Command Prefixes

Commands in both implicit and explicit rules can have prefixes that modify how MAKE treats the commands. The following table lists the prefixes you can use in makefiles:

Prefix	Description
@	Does not display the command while it's being executed.
-<num>	Stops processing commands in the makefile when the exit code returned from command exceeds the integer num. Normally, MAKE aborts if the exit code is nonzero. No space is allowed between - and <num>.
-	Continues processing commands in the makefile, regardless of the exit codes they return.

&	Expands either the macro $**, which represents all dependent files, or the macro $?, which represents all dependent files stamped later than the target. Execute the command once for each dependent file in the expanded macro.
!	Behaves like the & prefix.

Using @

The following command uses the @ prefix, which prevents MAKE from displaying the command onscreen:

```
diff.exe : diff.obj
   @bcc32 diff.obj
```

Using -<num> and —

The prefixes -<num> and - (hyphen) control the makefile processing when errors occur. You can choose to continue with the MAKE process if any error occurs, or you can specify a number of errors to tolerate.

In the following example, MAKE continues processing if BCC32 returns errors:

```
target.exe : target.obj
target.obj : target.cpp
      -bcc32 -c target.cpp
```

Using &

The & prefix issues a command once for each dependent file. It is especially useful for commands that don't take a list of files as parameters. For example:

```
copyall : file1.cpp file2.cpp
    &copy $** c:\temp
```

invokes COPY twice, as follows:

```
copy file1.cpp c:\temp
copy file2.cpp c:\temp
```

Without the & modifier, MAKE would call COPY only once. **Note:** the & prefix only works with $** and $! macros.

MAKE Command Operators

While you can use any operating system command in a MAKE command section, you can also use the following special operators:

Operator	Description
<	Use input from a specified file rather than from standard input.
>	Send the output from command to file.
>>	Append the output from command to file.
<<	Create a temporary inline file and use its contents as standard input to command. Also, create temporary response file when -N is used. Note: this is only for use with Microsoft's NMAKE.
&&	Create a temporary response file and insert its name in the makefile.
<delimiter>	Use delimiters with temporary response files. You can use any character other than # as a delimiter. Use << and && as a starting and ending delimiter for a temporary file. Any characters on the same line and immediately following the starting delimiter are ignored. The closing delimiter must be written on a line by itself.

See Also

Using MAKE.EXE (see page 183)

MAKE Macros (see page 191)

MAKE Directives (see page 186)

3.1.1.16 Message Options

Use the -w option to specify message options for the CodeGear C++ compiler:

- To enable a particular warning message, enter the -w option with a one- to three-letter option code.
- To disable the warning message, enter the -w- option with a one- to three- letter option code.

Examples

To display all warning and error messages, include the **-w** command option in your BCC32 command:

```
-w
```

To enable reporting of the warning "Mixing pointers to different 'char' types" (option code **ucp**, Default OFF), include the following flag in your BCC32 command:

```
-wucp
```

To disable reporting of the warning "Declaration ignored" (option code **dig**, Default ON), include the following flag in your BCC32 command:

```
-w-dig
```

To display help on the warning message codes, enter a compiler command that contains only the -h and -w options:

```
bcc32 -h -w
CodeGear C++ 5.92 for Win32 Copyright (c) 1993, 2007 CodeGear
Available options (* = default setting, xxx = has sub-options: use -h -X):
(Note: -X- or -w-XXX will usually undo whatever was set or unset by -X)
  -w      Display all warnings
  -w!     Return non-zero from compiler on warnings
  -wamb   8000 Ambiguous operators need parentheses
  -wamp   8001 Superfluous & with function
* -wasc   8002 Restarting compile using assembly
  -wasm   8003 Unknown assembler instruction
* -waus   8004 '%s' is assigned a value that is never used
* -wali   8086 Incorrect use of #pragma alias "aliasName"="substituteName"
* -watr   8100 '%s' attribute directive ignored
  -wbbf   8005 Bit fields must be signed or unsigned int
* -wbei   8006 Initializing %s with %s
* -wbig   8007 Hexadecimal value contains too many digits
* -wccc   8008 Condition is always %s
  -wcln   8009 Constant is long
* -wcom   8010 Continuation character \ found in // comment
* -wcpt   8011 Nonportable pointer comparison
* -wcsu   8012 Comparing signed and unsigned values
* -wcod   8093 Incorrect use of #pragma codeseg [seg_name] ["seg_class"] [grou

  -wdef   8013 Possible use of '%s' before definition
* -wdig   8014 Declaration ignored
* -wdpu   8015 Declare '%s' prior to use in prototype
* -wdsz   8016 Array size for 'delete' ignored
* -wdup   8017 Redefinition of '%s' is not identical
* -weas   8018 Assigning %s to %s
* -weff   8019 Code has no effect
* -wext   8020 '%s' is declared as both external and static
  -wexc   8101 Extern C linkage ignored
* -whch   8021 Handler for '%s' hidden by previous handler for '%s'
* -whid   8022 '%s' hides virtual function '%s'
* -wias   8023 Array variable '%s' is near
* -wibc   8024 Base class '%s' is also a base class of '%s'
* -will   8025 Ill-formed pragma
  -winl   8026 Functions %s are not expanded inline
  -winl   8027 Functions containing %s are not expanded inline
```

```
* -wifr   8085 Function '%s' redefined as non-inline
  -wimp   8102 Implicit conversion of '%s' to '%s'
* -wlin   8028 Temporary used to initialize '%s'
* -wlvc   8029 Temporary used for parameter '%s'
* -wlvc   8030 Temporary used for parameter '%s' in call to '%s'
* -wlvc   8031 Temporary used for parameter %d
* -wlvc   8032 Temporary used for parameter %d in call to '%s'
* -wmpc   8033 Conversion to '%s' will fail for members of virtual base '%s'
* -wmpd   8034 Maximum precision used for member pointer type '%s'
* -wmsg   8035 %s
* -wmes   8095 Incorrect use of #pragma message( "string" )
* -wmcs   8096 Incorrect use of #pragma code_seg(["seg_name"[,"seg_class"]])
* -wmcc   8098 Multi-character character constant
  -wnak   8036 Non-ANSI keyword used: '%s'
  -wnak   8036 Non-ANSI keyword used: '%s'
* -wncf   8037 Non-const function %s called for const object
* -wnci   8038 Constant member '%s' is not initialized
* -wncl   8039 Constructor initializer list ignored
* -wnfd   8040 Function body ignored
* -wngu   8041 Negating unsigned value
* -wnin   8042 Initializer for object '%s' ignored
* -wnma   8043 Macro definition ignored
* -wnmu   8044 #undef directive ignored
  -wnod   8045 No declaration for function '%s'
* -wnop   8046 Pragma option pop with no matching option push
* -wnsf   8047 Declaration of static function '%s(...)' ignored
* -wnst   8048 Use qualified name to access member type '%s'
* -wntd   8049 Use '< <' for nested templates instead of '<<'
* -wnto   8050 No type OBJ file present.  Disabling external types option.
* -wnvf   8051 Non-volatile function %s called for volatile object
* -wnpp   8083 Pragma pack pop with no matching pack push
* -wobi   8052 Base initialization without a class name is now obsolete
* -wobs   8053 '%s' is obsolete
* -wofp   8054 Style of function definition is now obsolete
* -wosh   8055 Possible overflow in shift operation
* -wovf   8056 Integer arithmetic overflow
* -wonr   8097 Not all options can be restored at this time
* -wpar   8057 Parameter '%s' is never used
* -wpch   8058 Cannot create pre-compiled header: %s
* -wpck   8059 Structure packing size has changed
* -wpia   8060 Possibly incorrect assignment
  -wpin   8061 Initialization is only partially bracketed
* -wpow   8062 Previous options and warnings not restored
* -wpre   8063 Overloaded prefix 'operator %s' used as a postfix operator
* -wpro   8064 Call to function with no prototype
* -wpro   8065 Call to function '%s' with no prototype
  -wprc   8084 Suggest parentheses to clarify precedence
* -wpcm   8094 Incorrect use of #pragma comment( <type> [,"string"] )
* -wpsb   8099 Static `main' is not treated as an entry point
* -wrch   8066 Unreachable code
* -wret   8067 Both return and return with a value used
* -wrng   8068 Constant out of range in comparison
* -wrpt   8069 Nonportable pointer conversion
* -wrvl   8070 Function should return a value
  -wsig   8071 Conversion may lose significant digits
* -wspa   8072 Suspicious pointer arithmetic
  -wstu   8073 Undefined structure '%s'
  -wstv   8074 Structure passed by value
* -wsus   8075 Suspicious pointer conversion
  -wstl   8087 '%s::operator==' must be publicly visible to be contained by a '% s'
  -wstl   8089 '%s::operator<' must be publicly visible to be contained by a '%s'
  -wstl   8090 '%s::operator<' must be publicly visible to be used with '%s'
  -wstl   8091 %s argument %s passed to '%s' is a %s iterator: %s iterator required
  -wstl   8092 %s argument %s passed to '%s' is not an iterator: %s iterator required
* -wtai   8076 Template instance '%s' is already instantiated
* -wtes   8077 Explicitly specializing an explicitly specialized class member makes no sense
```

```
* -wthr   8078 Throw expression violates exception specification
  -wucp   8079 Mixing pointers to different 'char' types
  -wuse   8080 '%s' is declared but never used
* -wvoi   8081 void functions may not return a value
* -wzdi   8082 Division by zero
```

3.1.1.17 Module Definition Files

You use module definition files with ILINK32. A module definition file is an ASCII text file that provides information to ILINK32 about the contents and system requirements of a Windows application. Use IMPDEF to create a module definition file.

The module definition file names the .EXE or .DLL, identifies the application type, lists imported and exported functions, describes the code section and data segment attributes, lets you specify attributes for additional code sections and data segments, specifies the size of the stack, and provides for the inclusion of a stub program.

CODE Statement

Defines the default attributes of code sections. Code sections can have any name, but must belong to section classes whose name ends in CODE (such as CODE or MYCODE).

```
CODE [PRELOAD | LOADONCALL]
      [EXECUTEONLY | EXECUTEREAD]
```

- **PRELOAD** means code is loaded when the calling program is loaded.
- **LOADONCALL** (the default) means the code is loaded when called by the program.
- **EXECUTEONLY** means a code section can only be executed.
- **EXECUTEREAD** (the default) means the code section can be read and executed.
- **FIXED** (the default) means the section remains at a fixed memory location.
- **MOVEABLE** means the sction can be moved.
- **DISCARDABLE** means the section can be discarded if it is no longer needed (this implies MOVEABLE).
- **NONDISCARDABLE** (the default) means the section can not be discarded.

DATA Statement

Defines attributes of data segments:

```
DATA [NONE | SINGLE | MULTIPLE]
     [READONLY | READWRITE]
     [PRELOAD | LOADONCALL]
     [SHARED | NONSHARED]
```

- **NONE** means that there is no data section created. This option is available only for libraries.
- **SINGLE** (the default for .DLLs) means a single data section is created and shared by all processes.
- **MULTIPLE** (the default for .EXEs) means that a data section is created for each process.
- **READONLY** means the data section can be read only.
- **READWRITE** (the default) means the data section can be read and written to.
- **PRELOAD** means the data section is loaded when a module that uses it is first loaded.
- **LOADONCALL** (the default) means the data section is loaded when it is first accessed. (LOADONCALL is ignored for 32-bit applications.)
- **SHARED** means one copy of the data section is shared among all processes.
- **NONSHARED** (the default for programs .DLLs) means a copy of the data section is loaded for each process needing to use the data section.

DESCRIPTION Statement (optional)

Inserts text into the application module and is typically used to embed author, date, or copyright information:

```
DESCRIPTION 'Text'
```

Text is an ASCII string delimited with single quotes.

EXETYPE Statement

Defines the default executable file (.EXE) header type for applications. You can leave this section in for 32-bit applications for backward compatibility, but if you need to change the EXETYPE, see the NAME statement.

```
EXETYPE  [WINDOWAPI] | [WINDOWCOMPAT]
```

- **WINDOWAPI** is a Windows executable, and is equivalent to the ILINK32 option -aa.
- **WINDOWCOMPAT** is a Windows-compatible character-mode executable, and is equivalent to the ILINK32 option -ap.

EXPORTS Statement

Defines the names and attributes of functions to be exported. The EXPORTS keyword marks the beginning of the definitions. It can be followed by any number of export definitions, each on a separate line.

```
EXPORTS
   <EsportName> [<Ordinal>]
   [RESIDENTNAME] [<Parameter>]
```

- **<ExportName>** specifies an ASCII string that defines the symbol to be exported as follows: `<EntryName> [=<InternalName>]` **<EntryName>** is the name listed in the executable file's entry table and is externally visible. **<InternalName>** is the name used within the application to refer to this entry.
- **<Ordinal>** defines the function's ordinal value as follows: `@<ordinal>` where `<ordinal>` is an integer value that specifies the function's ordinal value. When an application or DLL module calls a function exported from a DLL, the calling module can refer to the function by name or by ordinal value. It is faster to refer to the function by ordinal because string comparisons aren't required to locate the function. To use less memory, export a function by ordinal (from the point of view of that function's DLL) and import/call a function by ordinal (from the point of view of the calling module). When a function is exported by ordinal, the name resides in the nonresident name table. When a function is exported by name, the name resides in the resident name table. The resident name table for a module is in memory whenever the module is loaded; the nonresident name table isn't.
- **RESIDENTNAME** specifies that the function's name must be resident at all times. This is useful only when exporting by ordinal (when the name wouldn't be resident by default).
- **<Parameter>** is an optional integer value that specifies the number of words the function expects to be passed as parameters.

HEAPSIZE Statement

Defines the number of bytes the application needs for its local heap. An application uses the local heap whenever it allocates local memory.

```
HEAPSIZE <Reserve>[, <Commit>]
```

- **<Reserve>** can be a decimal or hex value, the default of which is 1MB. To help with backward (16-bit) compatibility for 32-bit ports, the linker uses the default value of 1MB if you specify in the .DEF ile a reserve value less than 64K.
- **<Commit>** is a decimal or hex value. The commit size is optional, and if not specified defaults to 4K. The minimum commit size you can specify is 0. In addition, the specified or default commit size must always be smaller or equal to the reserve size.

Reserved memory refers to the maximum amount of memory that can be allocated either in physical memory or in the paging file. In other words, reserved memory specifies the maximum possible heap size. The operating system guarantees that the specified amount of memory will be reserved and, if necessary, allocated.

The meaning of committed memory varies among operating systems. In Windows NT, commited memory refers to the amount of physical memory allocated for the heap at application load/initialization time. Committed memory causes space to be allocated either in physical memory or in the paging file. A higher commit value saves time when the application needs more heap space, but increases memory requirements and possible startup time.

You can override any heap reserve or commit size specified in the .DEF file with the -H or -Hc ILINK32 command-line options. -H lets you specify a heap reserve size less than the 64K minimum allowed in the .DEF file.

IMPORTS Statement

Defines the names and attributes of functions to be imported from DLLs. The IMPORTS keyword marks the beginning of the definitions followed by any number of import definitions, each on a separate line.

```
IMPORTS [<InternalName>=]<ModuleName>.<Entry>
```

- **<InternalName>** is an ASCII string that specifies the unique name the application uses to call the function.
- **<ModuleName>** specifies one or more uppercase ASCII characters that define the name of the executable module containing the function. The module name must match the name of the executable file. For example, the file SAMPLE.DLL has the module name SAMPLE.
- **<Entry>** specifies the function to be imported--either an ASCII string that names the function or an integer that gives the function's ordinal value.

Instead of listing imported DLL functions in the IMPORTS statement, you can either specify an import library for the DLL in the ILINK32 command line, or include the import library for the DLL in the project manager in the IDE.

You must use either __declspec(dllimport) or __import to import any function, class, or data you want imported. The prefered method is to use __declspec(dllimport)

The IMPORTS keyword can also be applied to (global) variables, as follows:

```
[ in the dll source file (dll.cpp) ] int sample = 100;

[ dll.def ]     EXPORTS       _sample

[ in application source file (app.cpp) ]
int _declspec(dllimport) sample;

[ app.def ]
  IMPORTS
    dll._sample
```

LIBRARY Statement

Defines the name of a DLL module. A module definition file can contain either a LIBRARY statement to indicate a .DLL or a NAME statement to indicate a .EXE. A library's module name must match the name of the executable file. For example, the library MYLIB.DLL has the module name MYLIB.

```
LIBRARY <LibrarName> [INITGLOBAL | INITINSTANCE]
```

- **<LibraryName>** (optional) is an ASCII string that defines the name of the library module. If you don't include a LibraryName, ILINK32 uses the file name with the extension removed. If the module definition file includes neither a NAME nor a LIBRARY statement, ILINK32 assumes a NAME statement without a ModuleName parameter.
- **INITGLOBAL** means the library-initialization routine is called.
- **INITINSTANCE** means the library-initialization routine is called each time a new process uses the library.

NAME Statement

Is the name of the application's executable module. The module name identifies the module when exporting functions. NAME must appear before EXETYPE. If NAME and EXETYPE don't specify the same target type, the linker uses the type listed with NAME.

```
NAME <ModuleName> [WINDOWSAPI] | [WINDOWCOMPAT]
```

- **<ModuleName>** (optional) specifiesone or more uppercase ASCII characters that name the executable module. The name must match the name of the executable file. For example, an application with the executable file SAMPLE.EXE has the module name SAMPLE. If <ModuleName> is missing, ILINK32 assumes that the module name matches the file name of the executable file. For example, if you do not specify a module name and the executable file is named MYAPP.EXE, ILINK32 assumes that the module name is MYAPP. If the module definition file includes neither a NAME nor a LIBRARY statement, ILINK32 assumes a NAME statement without a <ModuleName> parameter.

- **WINDOWSAPI** specifies a Windows executable, and is equivalent to the ILINK32 option **-aa**.
- **WINDOWCOMPAT** specifies a Windows-compatible character-mode executable, and is equivalent to the ILINK32 option **-ap**.

SECTIONS Statement

Lets you set attributes for one or more section in the image file. you can use this You can use this statement to override the default attributes for each different type of section.

```
SECTIONS<section_name> [CLASS '<ClassName>'] <attributes>
```

- **SECTIONS** marks the beginning of a list of section definitions.
- After the SECTIONS keyword, each section definition must be listed on a separate line. Note that the SECTIONS keyword can be on the same line as the first definition or on a preceding line. In addition, the .DEF file can contain one or more SECTIONS statements. The SEGMENTS keyword is supported as a synonym for SECTIONS.
- **<ClassName>** is case sensitive. The CLASS keyword is supported for compatibility but is ignored.
- **<Attributes>** is one or more of the following: EXECUTE, READ, SHARED, and WRITE.

SEGMENTS Statement

Defines the attributes of additional code and data sections. The SEGMENTS keyword is supported as a synonym for SECTIONS. The syntax is:

```
SEGMENTS

   <SegmentName> [CLASS '<ClassName>']

        [<MinAlloc>]
        [SHARED | NONSHARED]
        [PRELOAD | LOADONCALL]
```

- **<SegmentName>** is a character string that names the new segment. It can be any name, including the standard segment names _TEXT and _DATA, which represent the standard code and data segments.
- **<ClassName>** (optional) is the class name of the specified segment. If no class name is specified, ILINK32 uses the class name CODE.
- **<MinAlloc>** (optional) is an integer that specifies the minimum allocation size for the segment. ILINK32 ignores this value.
- **SHARED** means one copy of the segment is shared among all processes.
- **NONSHARED** (the default for .EXEs and .DLLs) means a copy of the segment is loaded for each process needing to use the data segment.
- **PRELOAD** means that the segment is loaded immediately.
- **LOADONCALL** means that the segment is loaded when it is accessed or called (this is ignored by ILINK32). The Resource Compiler may override the LOADONCALL option and preload segments instead.

STACKSIZE Statement

Defines the number of bytes the application needs for its local stack. An application uses the local stack whenever it makes function calls.

```
STACKSIZE <Reserve>[, <Commit>]
```

- **<Reserve>** can be a decimal or hex value, the default of which is 1MB. To help with backward (16-bit) compatibility, the linker uses the default value of 1MB if you specify in the .DEF file a reserve value less than 64K.
- **<Commit>** is a decimal or hex value. The commit size is optional, and if not specified defaults to 8K. The minimum commit size you can specify is 4K. In addition, the specified or default commit size must always be smaller or equal to the reserve size.

Reserved memory refers to the maximum amount of memory that can be allocated either in physical memory or in the paging file. In other words, reserved memory specifies the maximum possible stack size.

The operating system guarantees that the specified amount of memory will be reserved and, if necessary, allocated.

The meaning of committed memory varies among operating systems. In Windows NT, commited memory refers to the amount of physical memory allocated for the stack at application load/initialization time. Committed memory causes space to be allocated either in physical memory or in the paging file. A higher commit value saves time when the application needs more stack space, but increases memory requirements and possible startup time.

You can override any stack reserve or commit size specified in the .DEF file with the ILINK32 -S or -Sc command-line options. -S lets you specify a stack reserve size less than the 64K minimum allowed in the .DEF file.

Note: Do not use the STACKSIZE statement when compiling .DLLs.

STUB Statement

Appends a DOS executable file specified by FileName to the beginning of the module. The executable stub displays a warning message and terminates if the user attempts to run the executable stub in the wrong environment (running a Windows application under DOS, for example).

```
STUB '<FileName>'
```

- **<FileName>** is the name of the DOS executable file to be appended to the module. The name must have the DOS file name format. If the file named by FileName is not in the current irectory, ILINK32 searches for the file in the directories specified by the PATH environment variable.

C++Builder adds a built-in stub to the beginning of a Windows application unless a different stub is specified with the STUB statement. You should not use the STUB statement to include WINSTUB.EXE because the linker does this automatically.

SUBSYSTEM Statement

Lets you specify the Windows subsystem and subsystem version number for the application being linked.

```
SUBSYSTEM [<subsystem>,]<subsystemID>
```

- **<Subsystem>** (optional) can be any one of the following values: WINDOWS, WINDOWAPI, WINDOWCOMPAT. If you do not specify a subsystem, the linker defaults to a WINDOWS subsystem.
- **<SubsystemID>** must use the format d.d where d is a decimal number. For example, if you want to specify Windows 4.0, you could use either one of the following two SUBSYSTEM statements: SUBSYSTEM 4.0 SUBSYSTEM WINDOWS 4.0

You can override any SUBSYSTEM statement in a .DEF file using the -a and -V ILINK32 command-line options.

See Also

ILINK32 (see page 174)

IMPDEF (see page 179)

3.1.1.18 Using Precompiled Header Files

Precompiled header files can dramatically increase compilation speed by storing an image of the symbol table on disk in a file, then later reloading that file from disk instead of parsing all the header files again. Directly loading the symbol table from disk is much faster than parsing the text of header files, especially if several source files include the same header file.

To use precompiled header files, specify the various -H options in your BCC32 command.

Precompiled Header Options

Compiler Option	Description	Details
-H	Generate and use	When you enable this option, the compiler generates and uses precompiled headers. The default file name is BC32DEF.CSM for the command-line compiler.
-Hu	Use but do not generate	When you enable this option, the compiler uses preexisting precompiled header files; new precompiled header files are not generated.

H	Do not generate or use	When you enable this option, the compiler does not generate or use precompiled headers. Default = Do not generate or use (-H-)
-Hc	Cache precompiled headers	When you enable this option, the compiler caches the precompiled headers it generates. This is useful when you are precompiling more than one header file. To use this option, you must also enable the Generate and Use (-H) precompiled header option. Default = Off.
-He	Generate external type files (precompiled headers)	When you enable this option, the compiler generates a file or files that contain debug type information for all the symbols contained in the precompiled headers. The filenames end with .#xx extension, where xx is 00 for the first file generated and is incremented for each additional type-information file required. Using this option dramatically decreases the size of your .OBJ files, because debug type information is centralized and is not duplicated in each .OBJ file. Default = On.
-H=<filename>	Precompiled header filename	Use this option to specify the name of your precompiled header file. When you set this option, the compiler generates and uses the precompiled header file that you specify.

-Hh=<xxx> or -H\"<xxx>\"	Stop precompiling after header file option	This option terminates compiling the precompiled header after processing the file specified as xxx. You can use this option to reduce the disk space required for precompiled headers.
		The -Hh syntax is generally easier to use than the -H syntax. Examples:
		`-Hh=myhdr.h`
		`-H\"myhdr.h\"`
		`-Hh="C:\Program Files\myhdr.h"`
		`-H\"C:\Program Files\myhdr.h\"`
		When you use this option, the file you specify must be included from a source file for the compiler to generate a precompiled header file.
		You can also use #pragma hdrstop in your source file to specify when to stop the generation of precompiled headers.
		You cannot specify a header file that is included from another header file. For example, you cannot list a header included by windows.h because doing this would cause the precompiled header file to be closed before the compilation of windows.h was completed.

See Also

ILINK32.EXE (see page 174)

3.1.1.19 RLINK32.DLL, the Resource Linker (C++)

RLINK32.DLL is the resource linker that binds resources, in .RES file form, to an .EXE file, and marks the resulting .EXE file as a Windows executable. RLINK32.DLL also:

- Links the resources by fixing up string tables and message tables and then binding these linked resources into the executable.
- Is called by ILINK32 and is used for 32-bit resources.

See Also

BRC32.EXE (see page 163)

ILINK32.EXE (see page 174)

3.1.1.20 TDUMP.EXE, the File Dumping Utility

TDUMP.EXE produces a file dump that shows the structure of a file.

TDUMP breaks apart a file structurally and uses the file's extension to determine the output display format. TDUMP recognizes many file formats, including .EXE, .OBJ, and .LIB files. If TDUMP doesn't recognize an extension, it produces a hexadecimal dump of the file. You can control the output format by using the TDUMP command-line options when you start the program.

TDUMP's ability to peek at a file's inner structure displays not only a file's contents, but also how a file is constructed. Moreover, because TDUMP verifies that a file's structure matches its extension, you can also use TDUMP to test file integrity.

Command Line Syntax

```
TDUMP [<options>] <inputfile> [<listfile>] [<options>]
```

- **<inputfile>** is the file whose structure you want to display (or “dump”).
- **<listfile>** is an optional output filename (you can also use the standard DOS redirection command (>).
- **<options>** are any of the TDUMP command line options.

To display command line help, enter:

```
tdump
```

You can use either the / or the - switch character. For example, these two commands are equivalent:

```
TDUMP -e1 -v demo.exe
TDUMP /el /v demo.exe
```

TDUMP Options

Option	Description
-a, -a7	TDUMP automatically adjusts its output display according to the file type. You can force a file to be displayed as ASCII by including the -a or -a7 option. -a produces an ASCII file display, which shows the offset and the contents in displayable ASCII characters. A character that is not displayable (like a control character) appears as a period. -a7 converts high-ASCII characters to their low-ASCII equivalents. This is useful if the file you are dumping sets high-ASCII characters as flags (WordStar files do this).
-b#	Allows you to display information beginning at a specified offset. For example, if you want a dump of MYFILE starting from offset 100, you would use: TDUMP -b100 MYFILE
-C	Causes TDUMP to dump information found in COFF format files (.OBJ and .LIB). This option is useful when linking with Microsoft .OBJ and .LIB files.
-d	Causes TDUMP to dump any CodeGear 32-bit debug information found in the .OBJ file. If you do not specify this option, TDUMP displays raw data only.

-e, -el, -er, -ex	All four options force TDUMP to display the file as an executable (.EXE) file. An .EXE file display consists of information contained within a file that is used by the operating system when loading a file. If symbolic debugging information is present (Turbo Debugger or Microsoft CodeView), TDUMP displays it. TDUMP displays information for DOS executable files, NEW style executable files (Microsoft Windows and OS/2 .EXEs and DLLs), Linear Executable files, and Portable Executable (PE) files used by Windows NT and Windows 95/98 and Windows 2000. • **-e** Displays file as Executable (EXE/DLL, DOS, Win16, OS/2, PE) • **-ea[:v]** Displays All Exports unsorted, or (:v) sort by RVA (default dumps only named exports, sorting on the name) a[:v] Displays All Exports unsorted, or (:v) sort by RVA (default dumps only named exports, sorting on the name) • **-ed** Disables EXE debug information • **-ee[=x]** Lists Exports only from EXE/DLL (optional x lists matches only) • **-el** Suppresses line numbers in the display • **-eiID** Includes only .EXE table ID (HDR, OBJ, FIX, NAM, ENT) • **-em[=x]** Lists Imports only from EXE/DLL (optional x lists matches only) • **-em.[x]** Lists Imported modules only from EXE/DLL (optional x search string) • **-ep** Disables EXE PE header display • **-er** Prevents the relocation table from displaying • **-ex** Prevents the display of New style executable information. This means TDUMP only displays information for the DOS "stub" program
-h	Displays the dump file in hexadecimal (hex) format. Hex format consists of a column of offset numbers, 16 columns of hex numbers, and their ASCII equivalents (a period appears where no displayable ASCII character occurs). If TDUMP doesn't recognize the input file's extension, it displays the file in hex format (unless an option is used to indicate another format).
-i	Specifies the debug table(s) to use.

-l, -li, -le	Displays the output file in library (.LIB) file format. A library file is a collection of object files (see the -o option for more on object files). The library file dump displays library-specific information, object files, and records in the object file.
	The **-li** option tells TDUMP to display a short form of "impdef" records when dumping import libraries. You can also specify a search string using the following syntax:
	`-li=<string>`
	For example, this command:
	`tdump –li=codeptr import32.lib`
	results in the following output:
	`Impdef:(ord) KERNEL32.0336=ISBADCODEPTR`
	This output shows that the function is exported by ordinal, whose ordinal value is 336 (decimal). In addition, the output displays the module and function name.
	If you enter the following command:
	`tdump –li=walk import32.lib`
	TDUMP displays:
	`Impdef:(name) KERNEL32.????=HEAPWALK`
	This shows the output of a function exported by name.

-le	Tells TDUMP to display a short form of the "expdef" records when dumping OBJ files. You can also specify a search string using the following syntax: -le=<string>
-m	Leaves C++ names occurring in object files, executable files, and Turbo Debugger symbolic information files in "mangled" format. This option is helpful in determining how the C++ compiler "mangles" a given function name and its arguments.

o, -oc, -oi, -ox	• **-o** displays the file as an object (.OBJ) file. An object file display contains descriptions of the command records that pass commands and data to the linker, telling it how to create an .EXE file. The display format shows each record and its associated data on a record-by-record basis. • **-oc** causes TDUMP to perform a cyclic redundancy test (CRC) on each encountered record. The display differs from the -o display only if an erroneous CRC check is encountered (the TDUMP CRC value differs from the record's CRC byte). • **-oi<id>** includes only specified record types in the object module dump. Replace <id> with the name of the record to be displayed. For instance: `TDUMP -oiPUBDEF MYMODULE.OBJ` produces an object module display for MYMODULE.OBJ that displays only the PUBDEF records. The **-ox** and **-oi** options are helpful in finding errors that occur during linking. By examining the spelling and case of the EXTDEF symbol and the PUBDEF symbol, you can resolve many linking problems. For instance, if you receive an "unresolved external" message from the linker, use TDUMP -oiEXTDEF to display the external definitions occurring in the module causing the error. Then, use TDUMP -oiPUBDEF on the module containing the public symbol the linker could not match. Another use for the -oi switch is to check the names and sizes of the segments generated in a particular module. For instance: TDUMP -oiSEGDEF MYMODULE.OBJ displays the names, attributes, and sizes of all of the segments in MYMODULE. **Note:** To get a list of record types for -oi and -ox, use the command line options **-oi?** and **-ox?**.
-r	Causes TDUMP to display raw data.
-R	Causes TDUMP to dump relocation tables from 32-bit PE (Win32) format images. The default is to suppress these dumps.

-s,
-su,
-s[xx]
	Tells TDUMP to display strings. By default -s displays all strings in a given file. You can specify a string, or part of a string, using the following syntax:
	`-s=<string>`
	For example, the command:
	`TDUMP -s=black GoGame.exe`
	results in the following output of all strings containing 'black':
	56840: IDI_BLACK
	81965: Capture Black
	85038: black.bmp
	The optional string argument is case insensitive. To specify several words of a string, use quotes. For example:
	`TDUMP -s="game of go" GoGame.exe`
	The -su option is used in the same manner, with an optional string argument, and tells TDUMP to display Unix-style strings, meaning with no offset.
	For example, the command:
	`TDUMP -su=black GoGame.exe`
	results in the following output:
	IDI_BLACK
	Capture Black
	black.bmp
	For the -s[xx] switch, xx can be:
	• **#** - minimum string length to look for (default: 4)
	• **b#** - Beginning file offset (-b# is the same as -sb#)
	• **e#** - Ending file offset
	• **f** - Format (wrap) long strings
	• **s** - Search string with case sensitivity (default: no)
	• **u** - unix format: no offset (or decimal offsets, -h for hex)
	• **=x** - search string = x

-um	Displays mangled names as unmangled. You do not need to reformat the text to edit out only the mangled names before using this option. You can pass output that has mangled names anywhere in the text, and only the mangled names will be affected.
-v	Verbose display. If used with an .OBJ or .LIB file, TDUMP produces a hexadecimal dump of the record's contents without any comments about the records. If you use TDUMP on a Turbo Debugger symbol table, it displays the information tables in the order in which it encounters them. TDUMP doesn't combine information from several tables to give a more meaningful display on a per-module basis.
-x	Specifies the debug table(s) to exclude.

3.1.1.21 TLIB.EXE, the Library Manager

TLIB is a utility that manages libraries of .OBJ (object module) files. A library is a convenient way to deal with a collection of object modules as a unit.

The libraries included with the CodeGear C++ compiler were built with TLIB. You can use TLIB to build your own libraries, or to modify the CodeGear C++ libraries, your libraries, libraries furnished by other programmers, or commercial libraries you've purchased.

When TLIB modifies an existing library, TLIB creates a copy of the original library and gives it a .BAK extension.

You can use TLIB to:

- Create a new library from a group of object modules.
- Add object modules or other libraries to an existing library.
- Remove object modules from an existing library.
- Replace object modules from an existing library.
- Extract object modules from an existing library.
- List the contents of a new or existing library.

TLIB can also create (and include in the library file) an extended dictionary, which can be used to speed up linking.

Although TLIB is not essential for creating executable programs with the CodeGear C++ compiler, it is a useful programming productivity tool that can be indispensable for large development projects.

Command Line Syntax

```
TLIB <libname> [<options>] [<operations>] [@<respfile>] [, <listfile>]
```

To display command line help, enter:

```
tlib
```

Element	Description
<libname>	The DOS path name of the library you want to create or manage. Every TLIB command must be given a libname. Wildcards are not allowed. TLIB assumes an extension of .LIB if none is given. Use only the .LIB extension because both BCC32 and the IDE require the .LIB extension in order to recognize library files. Note: If the named library does not exist and there are add operations, TLIB creates the library.
<options>	TLIB accepts five general options (C, L, P, O, and 8) as well as five default action options (a, d, e, u, and x). See the headings immediately following this table for details about the TLIB <options> and about the default action options.
<operations>	Describes the actions you want TLIB to do and consists of a sequence of two-character operation codes. If you only want to examine the contents of the library, do not give any operations. For more information, see the heading later in this topic.
@<respfile>	The path and name of the response file you want to include. You can specify more than one response file. For more details, see the heading later in this topic.
<listfile>	The name of the text file that TLIB will produce to hold a listing of the library contents. The listfile name (if given), must be preceded by a comma. No listing is produced if you don't give a file name. The listing is an alphabetical list of each module. The entry for each module contains an alphabetical list of each public symbol defined in that module. The default extension for the listfile is .LST.

TLIB General Command Options

TLIB recognizes five <options>:

- /C (case-sensitive; see further information below)
- /l (produces a listing file on standard output)
- /P (page size; see further information below)
- /0 (zero; purges comment records)
- /8 (displays warnings and messages encoded as UTF8)

TLIB /C Option: Using Case-Sensitive Symbols in a Library

When you add a module to a library, TLIB maintains a dictionary of all public symbols defined in the modules of the library. All symbols in the library must be distinct. If you try to add a module to the library that duplicates a symbol, TLIB displays an error message and doesn't add the module.

Normally, when TLIB checks for duplicate symbols in the library, uppercase and lowercase letters are not treated differently (for example, the symbols lookup and LOOKUP are treated as duplicates). You can use the /C option to add a module to a library that includes symbols differing only in case.

Don't use /C if you plan to use the library with other linkers or let other people use the library.

TLIB normally rejects symbols that differ only in case because some linkers aren't case-sensitive. ILINK32 has no problem distinguishing uppercase and lowercase symbols. As long as you use your library only with ILINK32, you can use the TLIB /C option without any problems.

TLIB /P option: Setting the Page Size to Create a Large Library

The library page size determines the maximum combined size of all object modules in the library, which cannot exceed 65,536 pages. The default (and minimum) page size of 16 bytes allows a library of about 1 MB in size. To create a larger library, use the /P option to increase the page size. The page size must be a power of 2, and it cannot be smaller than 16 or larger than 32,768.

All modules in the library must start on a page boundary. For example, in a library with a page size of 32 (the lowest possible page size higher than the default 16), an average of 16 bytes will be lost per object module in padding. If you attempt to create a library that is too large for the given page size, TLIB will issue an error message and suggest that you use /P with the next available higher page size.

TLIB Default Action Options

The default action options allow you to specify an action or several actions that are used for every subsequent module that does not have an explicit command specified, or until a different action flag is set. Using the default action options enables you to specify modules in a TLIB command without having to prefix each module with a command or action symbol. The TLIB default action options perform actions similar to those performed by the TLIB operation flags described in **TLIB Operations List**.

TLIB Default Action Options

Symbol	Description
/a	Add the module to the library.
/d	Remove the module from the library.
/e	Extract the module without removing it.
/u	Replace the module in the library.
/x	Extract the module and remove it.

TLIB Operations List

The operation list describes the actions you want TLIB to do and consists of a sequence of operations given one after the other. Each operation consists of a one- or two-character action symbol followed by a file or module name.:

```
<symbol> modulename
```

You can put whitespace around either the action symbol or the file or module name, but not in the middle of a two-character action or in a name. You can put as many operations as you like on the command line, up to DOS's COMMAND.COM-imposed line-length limit of 127 characters. The order of the operations is not important. TLIB always applies the operations in a specific order:

To replace a module, remove it, then add the replacement module.

1. All extract operations are done first.
2. All remove operations are done next.
3. All add operations are done last.

TLIB finds the name of a module by stripping any drive, path, and extension information from the given file name.

Note that TLIB always assumes reasonable defaults. For example, to add a module that has an .OBJ extension from the current directory, you need to supply only the module name, not the path and .OBJ extension.

Wildcards are never allowed in file or module names.

TLIB recognizes three action symbols (*, +, -), which you can use singly or combined in pairs for a total of five distinct operations. The order of the characters is not important for operations that use a pair of characters.

To create a library, you add modules to a library that does not yet exist.

TLIB Action Symbols

Symbol	Name	Description
+	Add	TLIB adds the named file to the library. If the file has no extension, TLIB assumes an extension of .OBJ. If the file is itself a library (with a .LIB extension), then the operation adds all of the modules in the named library to the target library. If a module being added already exists, TLIB displays a message and does not add the new module.
—	Remove	TLIB removes the named module from the library. If the module does not exist in the library, TLIB displays a message. A remove operation needs only a module name. TLIB lets you enter a full path name with drive and extension included, but ignores everything except the module name.
*	Extract	TLIB creates the named file by copying the corresponding module from the library to the file. If the module does not exist, TLIB displays a message and does not create a file. If the named file already exists, it is overwritten. You can't directly rename modules in a library. To rename a module, extract and remove it, rename the file just created, then add it back into the library.
—* or *—	Extract and remove	TLIB copies the named module from the library to the file. If the module does not exist, TLIB displays a message and does not create a file. If the named file already exists, it is overwritten. TLIB removes the file name and then removes it from the library.
—+ or +—	Replace	TLIB replaces the named module with the corresponding file.

Using TLIB Response Files

A response file is an ASCII text file that contains all or part of a TLIB command. Using TLIB response files, you can build TLIB commands that are longer than one command line. If you need to perform a large number of operations or perform operations several times, response files can make your job easier.

Response files can:

- Contain more than one line of text; use the ampersand character (&) at the end of a line to indicate that another line follows.
- Include a partial list of commands. You can combine options from the the command line with options in a response file.
- Be used with other response files in a single TLIB command line.

TLIB Examples

These simple examples demonstrate some of the different things you can do with TLIB:

Example 1

To create a library named MYLIB.LIB with modules X.OBJ, Y.OBJ, and Z.OBJ, type:

```
tlib mylib +x +y +z
```

Example 2

To create a library named MYLIB.LIB and get a listing in MYLIB.LST too, type:

```
tlib mylib +x +y +z, mylib.lst
```

Example 3

To get a listing in CS.LST of an existing library CS.LIB, type:

```
tlib cs, cs.lst
```

Example 4

To replace module X.OBJ with a new copy, add A.OBJ and delete Z.OBJ from MYLIB.LIB, type:

```
tlib mylib -+x +a -z
```

Example 5

To extract module Y.OBJ from MYLIB.LIB and get a listing in MYLIB.LST, type:

```
tlib mylib *y, mylib.lst
```

Example 6

To create a new library named ALPHA, with modules A.OBJ, B.OBJ, ..., G.OBJ using a response file:

1. First create a text file, ALPHA.RSP, with: `+a.obj +b.obj +c.obj & +d.obj +e.obj +f.obj & +g.obj`
2. Then use the TLIB command, which produces a listing file named ALPHA.LST: `tlib alpha @alpha.rsp, alpha.lst`

Example 7

To update modules Y.OBJ and Z.OBJ and delete modules A.OBJ and B.OBJ from MYLIB.LIB:

```
tlib mylib /u Y.obj Z.obj /d A.obj B.obj
```

3.1.1.22 Using TOUCH.EXE

TOUCH.EXE updates a file's date stamp so that it reflects your system's current time and date.

Command Line Syntax

```
touch [<options>] <filename> [<filename>...]
```

If TOUCH cannot find a file that matches the name you specify, it creates a zero-length file with the correct date stamp. To suppress automatic file creation, use the -c option.

Because TOUCH is a 32-bit executable, it accepts long file names. In addition, you can use file names that contain the wildcard characters * and ? to "touch" more than a single file at a time. Use the -s option to update matching files in all subdirectories.

Before you use TOUCH, make sure your system's internal clock is set correctly.

TOUCH Options

TOUCH.EXE supports several command line options.

TOUCH Options

Option	Description
-c	Don't generate file if it doesn't already exist.
-d<mm>-<dd>-<yy>	Sets the date of the file to the specified date.
-r<filename>	Sets the time and date of file to match those of <filename>.
-h	Displays help information (same as typing TOUCH without options or filenames.

-s	Recurses through subdirectories.
t<hh>:<mm>:<ss>-	Sets the time of the file to the specified time.
-v	Verbose mode. Shows each file that has been TOUCHed.

See Also

Using MAKE (see page 183)

3.1.1.23 TRIGRAPH

Trigraphs are three-character sequences that replace certain characters used in the C language that are not available on some keyboards. Translating trigraphs in the compiler would slow compilation down considerably, so CodeGear C++ provides a filter named TRIGRAPH.EXE to handle trigraph sequences.

Command Line Syntax

```
trigraph [-u] <filename> [...]
```

TRIGRAPH Options

TRIGRAPH has three command options:

- **-nXXX** writes output files to the directory named in XXX.
- **-u** means to Undo insert trigraphs.
- **-xXXX** creates output files with the extension XXX.

The following table shows the trigraph sequences the TRIGRAPH recognizes:

TRIGRAPH Character Sequences

The following table shows the trigraph sequences the TRIGRAPH recognizes:

Trigraph	Character
??=	#
??(	[
??)	]
??/	\
??'	^
??<	{
??>	}
??-	~

3.1.1.24 RC.EXE, the Microsoft SDK Resource Compiler

RC is the command-line version of the standard Microsoft SDK resource compiler. It accepts a resource script file (.RC) as input and produces a resource object file (.RES) as output.

Both C++Builder 2009 and Delphi 2009 give you a choice of resource compilers. On the **Project ▸ Options ▸ Resource Compiler** dialog box, you can select either of the following:

- BRCC32.exe, the CodeGear resource compiler

- RC.exe, the Microsoft platform SDK Resource Compiler

RC supports Unicode characters in resource files and file names, as well as new Vista resource types such as icons with alpha channel.

The actual filename of the RC compiler in the RAD Studio environment is ERC.exe.

Differences Between BRCC32 and RC

If you choose to use RC, several differences between BRCC32 and RC need to be addressed in existing .RC files, as follows:

- To use Windows types or constants with RC, you need to explicitly `#include <winresrc.h>` for both C++ and Delphi.
- RC does not handle image data inline as BRCC32 does. To use an image with RC, you need to save the image and refer to the image inline.
- RC handles line continution differently than BRCC32. The simplest change is to combine strings with embedded newlines (\n).
- For RC, the order of command elements requires that all command options must come first, before the name of the resource file.
- RC does not allow trailing commas after the string in a STRINGLIST.
- RC treats strings as C-style strings. This means that you need to do either of the following:
- Escape a single backslash (\) in filenames by using a double backslash (\\).
- Use a forward slash (/) in place of a single backslash (\).
- For RC, specifying `#include <dirname>` does not search the current directory. To search the current directory, use `#include "dirname"`.
- To embed a double-quote character in a string, use two consecutive double-quote characters ("") instead of using the backslash escape character.
- RCDATA and byte values are handled differently. For example:
- BRCC32 treats '11 aa 22 bb' as a string of bytes.
- For RC, you would need to specify 0xaa11 0xbb22.

See Also

BRC32.EXE (see page 163)

Microsoft Resouce Compiler

Using RC Compiler to Build Resources

Compiling Resources with RC - From Digital Mars

Resource Compiler Options

3.1.1.25 WSDLIMP.EXE, the Command Line WSDL Import Tool

WSDLIMP generates code to represent the types and APIs that a WSDL document defines. This code can be used to write client applications that call on the Web Service that the WSDL (Web Services Description Language) document describes. If you want to write a server that implements the Web Service, one of the command line options tells the importer to generate implementation classes that you can then complete by filling in the bodies of the generated methods.

Note: WSDLIMP ignores any definitions in the WSDL document for which it can't generate code. That is, it can only import Web Services that use SOAP or document literal encoding.

```
Borland WSDLIMP Version 2.3 - $Rev: 16699 $
Copyright (c) 2008 Borland Software Corporation
Usage: WSDLIMP [options] <WSDL[File|URL]|UDDI BindingKey>
```

```
Code Generation Options:
 -C   Generate C++ code                    -P   Generate Pascal code
                        [ -option{+|-} default shown ]
 -Oa- Process optional/nillable elements  -Op+ Process inc/imported schemas
 -Od+ Generate Complex Type Destructors   -Oq- Quiet mode (Suppress Headers)
 -Of- Process Faults                      -Og- Use OLE GUIDs for interface
 -Oi- Ignore Schema errors                -Os- Generate Server skeleton code
 -Ok- Map pure collections to classes     -Ot- Output unused types
 -Ol- Generate Literal types              -Ou+ Unwrap Literal Parameters
 -Om- Allow out parameters                -Ov+ Verbose Info. in file
 -On- Declare Types in Namespace          -Ow+ Map strings to WideStrings
 -Oo+ One out param becomes return        -Ox+ Strong class Aliases
Other options:
 -D<path> Output directory path           -=+ Output filename after'=' in URL
 -U<url of UDDI Registry> UDDI Registry [NOTE: input must be UDDI bindingkey(s)

  @<Resp> Response file with list of WSDL|UDDIBindingKey to import

Proxy/Server Authentication:
-user:userName -pass:Password [-proxy:Proxy]
```

WSDLIMP.EXE imports a WSDL document, generating the code that lets your application call on the Web Service that the WSDL document describes. This is the command-line version of the Web Services importer, which is available on the Web Services tab of the new items dialog.

You can configure the output using the WSDLIMP command-line options when you start the program. These options let you specify whether the importer generates Object Pascal or C++ code, whether you want to generate implementation classes because you are writing a server, and how to handle various types of definitions in the WSDL document.

WSDLIMP Command Line Syntax

The command-line syntax for WSDLIMP is:

```
WSDLIMP [<options>] <Inputfile|URL>
```

<Inputfile> is a physical WSDL document to import.

<URL> is the URL for accessing a WSDL document that is published on the Web.

<options> stand for any of the WSDLIMP command-line options.

WSDLIMP Command Line Options

The following table lists the command-line options for WSDLIMP:

Options	Description
-C	Generate C++ code to represent the definitions in the WSDL file. You must use either this option or the –P option, but not both.
-P	Generate Object Pascal code to represent the definitions in the WSDL file. You must use either this option or the –C option, but not both.
-S	Generate implementation classes for the APIs that the WSDL document defines. This option is used when you want to write a server that implements the Web Service that the WSDL document describes.
-Oa	When the importer encounters a type that maps to a class with no members, change the imported code to an array. This option exists because in some cases when the WSDL document does not make consistent use of schema definitions, the importer has problems importing array types, which results in a type that is represented by a class with no members. This option is always safe to use unless the WSDL document describes a Web Service that uses document literal encoding. Document literal encoding can also give rise to an empty class when a method has no return value.
-Oi	Ignore ill-formed schemas when importing the WSDL document and attempt to import the Web Service anyway. This allows the importer to handle cases where the WSDL document is not completely well-formed but the intent is clear.

-Oo	Treat methods with a single output parameter as functions, mapping the single output parameter to the return value.
-Oq	Do not generate a header comment at the top of each generated file. The header comments indicate what options were used, list any warnings about the generated code, and so on.
-Oi	Generate type definitions for the two types that represent the input and output of a call when the Web Service uses document literal encoding. This option is only used with the –Ou+ option. When you do not specify –Ou+, the two type definitions are always generated.
-Ou+	Unwind the literal parameters of document literal encoding to generate method calls. When this option is selected, the importer converts the two types that
-Ov	Unwind the literal parameters of document literal encoding to generate method calls. When this option is selected, the importer converts the two types that
-Ov	Generate code comments (verbose information).
-R:<namelist>	Map strings in the WSDL document to the WideString type. WideStrings are more robust, because they can handle extended characters, but they are not as efficient as using AnsiStrings (the default).
-D<path>	Specifies the directory where WSDLIMP writes the files that it generates.
@<path>	Specifies a text file where each line lists a URL or WSDL file to import. You can optionally add '=OutputFileName' after the URL or file name to specify where to place the generated code for each imported WSDL document.

See Also

Using the Web Services Importer (see page 2302)

3.1.2 C++ Compiler Errors And Warnings (C++)

This section describes the RAD Studio C++ compiler error and warning messages.

Topics

Name	Description
E2066: Invalid MOM inheritance (C++) (see page 256)	The compiler issues this error if the currently compiled class doesn't have the same MOM (Microsoft Object Model) related flags set as its direct parent. This compiler error message is deprecated.
E2525: You must define _PCH_STATIC_CONST before including xstring to use this feature (C++) (see page 257)	You attempted to use a feature defined in xstring, part of the Dinkumware standard C++ library. The C++ compiler could not generate a precompiled header because there is a constant (defined in xstring) in the header. If you want to include xstring, you should first set the define _PCH_STATIC_CONST.
E2526: Property 'name' uses another property as getter/setter; Not allowed (C++) (see page 257)	Properties typically have both a getter and a setter, but a property cannot serve as either the getter or setter of another property.
E2008: Published property access functions must use __fastcall calling convention (C++) (see page 257)	The calling convention for access functions of a property (read, write, and stored) declared in a __published section must be __fastcall. This also applies to hoisted properties.
E2122: Function call terminated by unhandled exception 'value' at address 'addr' (C++) (see page 257)	This message is emitted when an expression you are evaluating while debugging includes a function call that terminates with an unhandled exception. For example, if in the debugger's evaluate dialog, you request an evaluation of the expression foo()+1 and the execution of the function foo() causes a GP fault, this evaluation produces the above error message. You may also see this message in the watches window because it also displays the results of evaluating an expression.
E2506: Explicit specialization of 'specifier' is ambiguous: must specify template arguments (C++) (see page 257)	In the following code, explicit template arguments are necessary:

E2483: Array dimension 'specifier' could not be determined (C++) (see page 258)	If, during instantiation of a type, an array dimension cannot be computed—usually this is due to some other error which would be reported—then this error will result. For example, if an array dimension is dependent upon a template parameter but an error occurs while it is being parsed and the template argument being substituted does not yield a legal constant expression, this error is displayed.
E2509: Value out of range (C++) (see page 258)	The inline assembler has detected a numeric overflow in one of your expressions. Make sure all of your numbers can fit in 32 bits.
E2510: Operand size mismatch (C++) (see page 258)	Help is not available for this item.
E2050: __declspec(delphireturn) class 'class' must have exactly one data member (C++) (see page 258)	This is an internal compiler error. A class marked as a delphireturn class has more than one non-static data member.
E2530: Unrecognized option, or no help available (C++) (see page 258)	You have entered a command element that the C++ compiler cannot parse, or the option you entered has no associated help. Try again.
E2527: Option 'name' cannot be set via 'name' (C++) (see page 258)	An attempt was made in a source file to set an option that cannot be set after either parsing or compiling of the file starts. Instead, set this option on the command line or in a `.cfg` file. For example, if a source file contains a `#pragma option push –v`, you need to remove the push or set /unset this option either on the command line or in a `.cfg` file.
E2528: Option 'name' must be set before compilation begins (C++) (see page 258)	An attempt was made in a source file to set an option that must be set before compiling starts. Instead, set this option on the command line, in a `.cfg` file, or at the top of the source file before the line `int foo();` .
E2074: Value after -g or -j should be between 0 and 255 inclusive (C++) (see page 259)	Both the -g and the -j command line options can be followed by an optional number. The compiler expects this number to be between 0 and 255 inclusive.
E2492: Properties may only be assigned using a simple statement, e.g. \"prop = value;\" (C++) (see page 259)	Assignments to properties should be made in simple assignment statements. If property assignments could become Lvalues, which happens when property assignments are embedded in larger statements, the getter is called to create the Lvalue, with all the side effects that getter causes. The compiler allows only one call to either the getter or the setter in a statement. For example:
E2505: Explicit instantiation requires an elaborated type specifier (i.e.,"class foo<int>") (C++) (see page 259)	The following code is illegal:
E2100: Invalid template declarator list (C++) (see page 259)	It is illegal for a declarator list to follow a template class declaration. For example:
E2102: Cannot use template 'template' without specifying specialization parameters (C++) (see page 260)	The generic form of a template must be referenced using specialization parameters. For example, for a template class named foo, taking two template parameters, then a legal reference might have the form
E2107: Invalid use of template 'template' (C++) (see page 260)	This error results when attempting to use a template template parameter in any way other than to reference a template specialization, or to pass that parameter in turn as a template template argument to another template. For example:
E2105: 'template' qualifier must specify a member template name (C++) (see page 260)	When parsing code that depends in some way upon a template parameter, it is sometimes impossible to know whether a member name will resolve to a template function name, or a regular parameter. In the following code, a 'template' qualifier is required in order to know if the '<' (less-then) operator should be parsed as the beginning character of a template argument list, or as a regular less-than operator:
E2066: Information not available (C++) (see page 261)	Help is not available for this item.
E2471: pragma checkoption failed: options are not as expected (C++) (see page 261)	You can use #pragma checkoption to check that certain switches are in the state that you expect. If #pragma checkoption detects that a switch is not in the expected state, the compiler displays this error. You can use the following syntax:
E2504: 'dynamic' can only be used with non-template member functions (C++) (see page 261)	You tried to use dynamic with a template member function. Dynamic functions are allowed for classes derived from TObject. Dynamic functions occupy a slot in every object that defines them, not in any descendants. That is, dynamic functions are virtual functions stored in sparse virtual tables. If you call a dynamic function, and that function is not defined in your object, the virtual tables of its ancestors are searched until the function is found.
E2191: '__far16' may only be used with '__pascal' or '__cdecl' (C++) (see page 261)	This is an internal compiler error. The compiler emits this message if the keyword __far16 is mixed with one of the keywords __pascal or __cdecl, all in the same declaration.
E2199: Template friend function 'function' must be previously declared (C++) (see page 262)	Not used

E2502: Error resolving #import: problem (C++) (see page 262)	Where `problem` can be any of the following relating to problems with the various attributes of #include (see page 692): unexpected import directive value attribute 'attribute'A value was supplied for the indicated attribute. None was expected. missing ')' in import directive attribute 'attribute'The value for the indicated attribute was incorrectly specified : a closing parenthesis is missing. unrecognized import directive attribute 'attribute'The indicated token is not a legitimate attribute for the #import directive. invalid values for raw_property_prefixes attributeThe values for the raw_property_prefixes attribute were incorrectly specified. unexpected duplicate property 'property'The indicated #import attribute was specified more than once -- this is an... more (see page 262)
E2501: Unable to open import file 'filename' (C++) (see page 262)	This error occurs when you use:
E2494: Unrecognized __declspec modifier (C++) (see page 262)	A _declspec modifier was given that is not valid.
E2493: Invalid GUID string (C++) (see page 263)	The GUID string does not have the form of a Globally Unique Identifier.
E2499: Invalid __declspec(uuid(GuidString)) format (C++) (see page 263)	This error happens when you used the wrong format to define your GuidString. GUIDs are defined for structs in the following way:
E2496: Invalid call to uuidof(struct type|variable) (C++) (see page 263)	The uuidof operator was given an incorrect argument.
E2511: Unterminated macro argument (C++) (see page 263)	A macro argument that was started on the line listed has not been properly terminated
E2489: Maximum option context replay depth exceeded; check for recursion (C++) (see page 263)	If this error is triggered, it means that recursive template instantiation has gone too deep. Check for compile-time recursion in your program, and limit it to no more than 256 levels.
E2488: Maximum token reply depth exceeded; check for recursion (C++) (see page 263)	If this error is triggered, it means that recursive template instantiation has gone too deep. Check for compile-time recursion in your program, and limit it to no more than 256 levels.
E2491: Maximum VIRDEF count exceeded; check for recursion (C++) (see page 263)	Too many VIRDEF symbols were allocated. The compiler imposes a limit to the number of VIRDEFs allowed per translation unit. Currently this limit is in the order of 16384 VIRDEFs. One way this could happen is if a program has more than 16384 functions.
E2230: In-line data member initialization requires an integral constant expression (C++) (see page 264)	Static const class members, which are initialized in the body of the class, have to be initialized with a constant expression of integral type.
E2241: VCL style classes need virtual destructors (C++) (see page 264)	Destructors defined in VCL style classes have to be virtual.
E2524: Anonymous structs/unions not allowed to have anonymous members in C++ (C++) (see page 264)	The C++ compiler requires that the members of an anonymous struct or union be named.
E2246: x is not abstract public single inheritance class hierarchy with no data (C++) (see page 264)	Internal compiler error. In some cases, the compiler will enforce restrictions on a class hierarchy. In this case, the restrictions would be that all classes are abstract classes, and all classes only have one parent.
E2249: = expected (C++) (see page 264)	The compiler expected an equal sign in the position where the error was reported but there was none. This is usually a syntax error or typo.
E2267: First base must be VCL class (C++) (see page 264)	Internal compiler error. In some cases, the compiler will enforce restrictions on a class hierarchy. In this case, the restrictions would be that the first parent of a class is a Delphi style class.
E2472: Cannot declare a member function via instantiation (C++) (see page 265)	If a declaration within a template class acquires a function type through a type dependent on a template-parameter and this results in a declaration that does not use the syntactic form of a function declarator to have function type, the program is ill-formed. For example:
E2515: Cannot explicitly specialize a member of a generic template class (C++) (see page 265)	You are trying to make a generic template into a specialized member. For example, the following code is illegal:
E2474: 'function' cannot be declared as static or inline (C++) (see page 265)	You attempted to declare a symbol as static or inline and this type of symbol cannot be defined as static or inline. Certain functions, like 'main' and 'WinMain' cannot be declared static or inline. 'main' is the entrypoint of console applications, and 'WinMain' is the entry point of Windows applications. For example, this error is displayed in the following cases:
E2498: Need previously defined struct GUID (C++) (see page 266)	This happens when you use the __uuidof operator without including a header that defines the GUID struct. So the following program code would display this error:
E2295: Too many candidate template specializations from 'specifier' (C++) (see page 266)	When reference a class template specialization, it is possible that more than one possible candidate might result from a single reference. This can only really happen among class partial specializations, when more than one partial specialization is contending for a possible match:
E2475: 'function' cannot be a template function (C++) (see page 266)	Certain functions, like 'main' and 'WinMain' cannot be declared as a template function. 'main' is the entrypoint of console applications, and 'WinMain' is the entry point of Windows applications. For example:

E2299: Cannot generate template specialization from 'specifier' (C++) (see page 267)	This error will result if an attempt is made to reference a template class or function in a manner which yields no possible candidate specializations. For example:
E2300: Could not generate a specialization matching type for 'specifier' (C++) (see page 267)	This error is no longer generated by the compiler.
E2497: No GUID associated with type:'type' (C++) (see page 267)	A variable or type was used in a context requiring a GUID, but the type does not have a GUID associated with it. GUIDs are associated with types using _declspec (uuid(GUID)).
E2522: Non-const function 'function' called for const object (C++) (see page 267)	Data type mismatch resulting in an erroneous function call. The object of the call (a non-const function) should be a const object.
E2523: Non-volatile function 'name' called for volatile object (C++) (see page 267)	Data type mismatch. The error is the result of an erroneous function call. The object of the call (a non-volatile function) should be a volatile object.
E2513: Cannot emit RTTI for 'parameter' in 'function' (C++) (see page 267)	The compiler issues this error if it cannot generate RTTI information for the return type of a parameter. See Runtime type information for more information.
E2512: Cannot emit RTTI for return type of 'function' (C++) (see page 267)	The compiler issues this error if the it cannot generate RTTI information for the return type of a function. See Runtime type information for more information.
E2507: 'class' is not a direct base class of 'class' (C++) (see page 268)	The first type is not a direct base class of the second type. A direct base class refers to the immediate derivations of that class, and not the derivations of its subclasses.
E2529: Path 'path' exceeds maximum size of 'n' (C++) (see page 268)	In looking up include files, the C++ compiler has encountered a file whose path name contains more characters than are allowed in the Windows maximum. Rename the path to a shorter name.
E2495: Redefinition of uuid is not identical (C++) (see page 268)	GUID's attached to structs have to be the same across multiple declarations and definitions of the same struct. So the following example would cause this error:
E2500: __declspec(selectany) is only for initialized and externally visible variables (C++) (see page 268)	You cannot use __declspec(selectany) with static variables, unitialized variables, etc.
E2482: String constant expected (C++) (see page 268)	The compiler expected a string constant at this location but did not receive one. This error is no longer generated by the compiler.
E2481: Unexpected string constant (C++) (see page 268)	There are times when the compiler does not expect a string constant to appear in the source input. For example:
E2386: Cannot involve parameter 'parameter' in a complex partial specialization expression (C++) (see page 268)	When declaring or defining a template class partial specialization, it is illegal to involve any of the non-type template parameters in complex expressions. They may only be referenced by name. For example:
E2387: Partial specializations may not specialize dependent non-type parameters ('parameter') (C++) (see page 269)	A partial specialization may not use a template parameter in its specialization argument list which is dependent on another type parameter. For example:
E2388: Argument list of specialization cannot be identical to the parameter list of primary template (C++) (see page 269)	When declaring a partial specialization, its specialization argument list must differ in some way from its basic parameter list. For example: template<class T>
E2389: Mismatch in kind of substitution argument and template parameter 'parameter' (C++) (see page 269)	When referencing a template specialization, all type parameters must be satisfied using type arguments, all non-type parameters require non-type arguments, and all template template parameters require either a template name, or another template template argument. Mismatching these requirements in any way will trigger the above error. For example:
E2480: Cannot involve template parameters in complex partial specialization arguments (C++) (see page 269)	A partial specialization cannot reference other template parameters in a nonvalue argument expression, unless it is simply a direct reference to the template parameter. For example:
E2392: Template instance 'template' is already instantiated (C++) (see page 270)	There are two ways to trigger this error. If –A is enabled (ANSI compliant mode), then attempting to explicitly instantiate a template specialization which has already been instantiated (either implicitly or explicitly) will cause this error. Regardless of –A, attempting to explicitly specialize a template specialization which has already been either implicit or explicitly instantiated will always trigger this error. For example:
E2393: Cannot take the address of non-type, non-reference template parameter 'parameter' (C++) (see page 270)	A template parameter has no address, and is not associated with any real "object". Therefore, to take its address, or attempt to assign to it, has no meaning. For example:
E2399: Cannot reference template argument 'arg' in template class 'class' this way (C++) (see page 270)	The compiler no longer generates this error.
E2397: Template argument cannot have static or local linkage (C++) (see page 270)	Only integral constant expressions, and the address of global variables with external linkage, may be used as template arguments. For example:
E2485: Cannot use address of array element as non-type template argument (C++) (see page 271)	Non-type template arguments may only be of integral type, or the address of a global variable. They cannot be the address of an array element. For example:
E2402: Illegal base class type: formal type 'type' resolves to 'type' (C++) (see page 271)	When instantiating a template class definition, if it is found that a declared base class does not resolve to an accessible class type, this error will result. For example:
E2403: Dependent call specifier yields non-function 'name' (C++) (see page 271)	The compiler no longer generates this error.

E2404: Dependent type qualifier 'qualifier' has no member type named 'name' (C++) (see page 271)	If a template declaration references a member of a dependent type, it is only possible to alert the user to the non-existence of this member during type instantiation for a given set of template arguments. For example:
E2405: Dependent template reference 'identifier' yields non-template symbol (C++) (see page 271)	If a template specialization reference within a template declaration yields a reference to a non-template during type instantiation, the above error will result. For example:
E2406: Dependent type qualifier 'qualifier' is not a class or struct type (C++) (see page 272)	If a dependent name reference within a template declaration results in a non-struct member qualification at instantiation time, the above error will result. For example:
E2407: Dependent type qualifier 'qualifier' has no member symbol named 'name' (C++) (see page 272)	If a template declaration references a member of a dependent type, it is only possible to alert the user to the non-existence of this member during type instantiation for a given set of template arguments. For example:
E2408: Default values may be specified only in primary class template declarations (C++) (see page 272)	Template functions, and class partial specializations, may not use default expressions in their template parameter lists. Only primary template declarations may do this. For example:
E2409: Cannot find a valid specialization for 'specifier' (C++) (see page 272)	This error is no longer generated by the compiler.
E2410: Missing template parameters for friend template 'template' (C++) (see page 273)	If a friend template is declared, but no template parameters are specified, this error will result. For example:
E2486: Cannot use address of class member as non-type template argument (C++) (see page 273)	Non-type template arguments may only be of integral type, or the address of a global variable. They cannot be the address of a class member. For example:
E2411: Declaration of member function default parameters after a specialization has already been expanded (C++) (see page 273)	If a member function of a class template is declared, and then a specialization of that class implicitly instantiated, and later that member function defined with default parameters specified, the above error will result. For example:
E2412: Attempting to bind a member reference to a dependent type (C++) (see page 273)	The compiler no longer generates this error.
E2414: Destructors cannot be declared as template functions (C++) (see page 273)	Destructors cannot be templates. For example:
E2473: Invalid explicit specialization of 'specifier' (C++) (see page 274)	Attempting to explicitly specialize a static data member or any non-template will cause this error.
E2490: Specialization within template classes not yet implemented (C++) (see page 274)	Explicit and partial specialization of member template classes and functions within template classes and nested template classes, is not supported.
E2416: Invalid template function declaration (C++) (see page 274)	The compiler no longer generates this error.
E2417: Cannot specify template parameters in explicit specialization of 'specifier' (C++) (see page 274)	The compiler no longer generates this error.
E2418: Maximum instantiation depth exceeded; check for recursion (C++) (see page 274)	The compiler only supports 256 levels of instantiation before it will trigger this error. The main problem is in controlling stack depth, because the parser uses recursive functions to manage type instantiation. Here is an example that would produce such an error:
E2420: Explicit instantiation can only be used at global scope (C++) (see page 274)	Explicit instantiation cannot be specified at any level other than namespace or global scope. For example:
E2422: Argument kind mismatch in redeclaration of template parameter 'parameter' (C++) (see page 275)	If a template is declared at one point in the translation unit, and then redeclared with template parameters of a different kind at another location, this error will result. For example:
E2423: Explicit specialization or instantiation of non-existing template 'template' (C++) (see page 275)	Attempting to explicit specialize or instantiate a template which does not exist is clearly illegal. For example:
E2479: Cannot have both a template class and function named 'name' (C++) (see page 275)	No other function or type may have the same name as a template class. For example:
E2484: The name of template class 'class' cannot be overloaded (C++) (see page 275)	Attempting to declare a function that overrides the name of a template class will cause this error. For example:
E2426: Explicit specialization of 'specifier' requires 'template<>' declaration (C++) (see page 275)	According to the standard, explicit specialization of any template now always require the "template<>" declarator syntax. For example:
E2487: Cannot specify default function arguments for explicit specializations (C++) (see page 276)	An explicit specialization of a function may not declare default function arguments. For example:
E2427: 'main' cannot be a template function (C++) (see page 276)	'main' cannot be declared as a template function. 'main' is the entry point of a console application, and it should be declared as a regular __cdecl function. This error message should not occur because it has been replaced with another one (E2475).
E2429: Not a valid partial specialization of 'specifier' (C++) (see page 276)	Internal compiler error.
E2430: Number of template parameters does not match in redeclaration of 'specifier' (C++) (see page 276)	If a template is redeclared with a different number of template parameters, this error will result. For example:
E2477: Too few template parameters were declared for template 'template' (C++) (see page 276)	If a member declaration or definition occurs outside of a template class, and that outer declaration uses a different number of template parameters than the parent class, this error will result. For example:
E2478: Too many template parameters were declared for template 'template' (C++) (see page 277)	If a member declaration or definition occurs outside of a template class, and that outer declaration uses a different number of template parameters than the parent class, this error will result. For example:

E2431: Non-type template parameters cannot be of floating point, class, or void type (C++) (see page 277)	Non-type template parameters are restricted as to what type they may be. Floating point, class and void types are illegal. For example:
E2434: Template declaration missing template parameters ('template<...>') (C++) (see page 277)	In a context where at least one template parameter is clearly required, if none are found this error will result. For example:
E2435: Too many template parameter sets were specified (C++) (see page 277)	If a member template is being defined outside of its parent class, and too many template parameter sets are declared, this error will result. For example:
E2436: Default type for template template argument 'arg' does not name a primary template class (C++) (see page 277)	If a template template parameter is to have a default type, that type must either be a generic template class name, or another template template parameter.
E2437: 'typename' should be followed by a qualified, dependent type name (C++) (see page 278)	Whenever the "typename" keyword is used in a template declaration or definition, it should always name a dependent type. For example:
E2438: Template template arguments must name a class (C++) (see page 278)	A template template parameter must always declare a new class name. For example:
E2439: 'typename' is only allowed in template declarations (C++) (see page 278)	The "typename" keyword must only be used within template declarations and definitions.
E2440: Cannot generate specialization from 'specifier' because that type is not yet defined (C++) (see page 278)	The compiler no longer generates this error.
E2441: Instantiating 'specifier' (C++) (see page 278)	Whenever a compiler error occurs while instantiating a template type, the context of what was being instantiated at that point in time will be reported to the user, in order to aid in detection of the problem.
E2503: Missing or incorrect version of TypeLibImport.dll (C++) (see page 279)	This error occurs when the compiler is trying to access TypeLibImport.dll but it either can't find it, it was corrupted, or you have the wrong version of it installed on your computer. You can reinstall it from the product CD.
E2470: Need to include header <typeinfo> to use typeid (C++) (see page 279)	When you use the 'typeid' function, you have to include the <typeinfo> header, otherwise you will get syntax errors. For example, consider a test case with the following code:
E2514: Cannot (yet) use member overload resolution during template instantiation (C++) (see page 279)	You are trying to overload a member during template instantiation. You cannot have calls to overloaded constant functions within array bounds initializers, for example.
E2508: 'using' cannot refer to a template specialization (C++) (see page 279)	The `using` keyword cannot refer to a template specialization.
E2462: 'virtual' can only be used with non-template member functions (C++) (see page 279)	The 'virtual' keyword can only be applied to regular member functions, not to member template functions. Consider a test case with the following code:
W8086: Incorrect use of #pragma alias "aliasName"="substituteName" (C++) (see page 280)	The directive #pragma alias is used to tell the linker that two identifier names are equivalent. You must put the two names in quotes. You will receive this warning if you don't use pragma alias correctly. For example, the following two lines both generate this warning:
W8099: Static main is not treated as an entry point (C++) (see page 280)	The main function has been created as static, and as such cannot be used as a valid entry point. Consider:
W8093: Incorrect use of #pragma codeseg [seg_name] ["seg_class"] [group] (C++) (see page 280)	The #pragma codeseg directive can be used to set or reset the name, class, and group of a segment. You have to follow the exact syntax mentioned in the warning message, and all names are optional. So these are all legal:
W8094: Incorrect use of #pragma comment(<type> [,"string"]) (C++) (see page 280)	The directive #pragma comment can be used to emit linker comment records. In this message, <type> can be any of the following: • user • lib • exestr • linker The type should be there but the string is optional.
W8085: Function 'function' redefined as non-inline (C++) (see page 281)	This warning is used to indicate when a certain function, which has been declared inline in one location, is redefined in another location to be non-inline.
W8105: %s member '%s' in class without constructors (C++) (see page 281)	A class that contains constant or reference members (or both) must have at least one user-defined constructor. Otherwise, there would be no way to ever initialize such members.
W8095: Incorrect use of #pragma message("string") (C++) (see page 281)	You can use pragma message to emit a message to the command line or to the message window. You would get this warning if you use the incorrect syntax, so
W8098: Multi-character character constant (C++) (see page 281)	This warning is issued when the compiler detects a multi-character integer constant, such as:

W8096: Incorrect use of #pragma code_seg(["seg_name"[,"seg_class"]]) (C++) (see page 281)	Pragma code_seg is similar to pragma codeseg, but with this one you can only modify the name and the class of a code segment. If you use the wrong syntax, you get this warning. The following examples show the correct usage:
W8083: Pragma pack pop with no matching pack push (C++) (see page 282)	Each #pragma pack(pop) should have a matching preceding #pragma pack(push) in the same translation unit. Pairs of 'push' and 'pop' can be nested. For example:
W8097: Not all options can be restored at this time (C++) (see page 282)	Your program has a #pragma pop at a place where it can't restore options. For example:
W8084: Suggest parentheses to clarify precedence (C++) (see page 282)	This warning indicates that several operators used in one expression might cause confusion about the applicable operator precedence rules. The warning helps create code that is more easy to understand and potentially less ambiguous. For example, compile the following code using the `-w` command line option:
W8092: 'type' argument 'specifier' passed to 'function' is not an iterator: 'type' iterator required (C++) (see page 282)	An argument that is not an iterator is being used with an STL algorithm that requires an iterator.
W8087: 'operator::operator==' must be publicly visible to be contained by a 'type' (C++) (see page 283)	A type that is being used with an STL container has a private 'operator=='.
W8090: 'type::operator<' must be publicly visible to be used with 'type' (C++) (see page 283)	A type that is being used with an STL container has a private 'operator<'. The type you're trying to use must be made public.
W8089: 'type::operator<' must be publicly visible to be contained by a 'type' (C++) (see page 283)	The type that is being used for an STL container has a private 'operator<'. The type that is being contained (type::operator) must be a public type. For example, if you were trying to instantiate a class type "vector<blah>", the error would be:
W8091: 'type' argument 'specifier' passed to 'function' is a 'iterator category' iterator: 'iterator category' iterator required (C++) (see page 283)	An incorrect iterator category is being used with an STL algorithm.
W8076: Template instance 'specifier' is already instantiated (C++) (see page 283)	You are trying to explicitly instantiate a template that was already implicitly instantiated. If –A is not enabled and an attempt is made to explicitly instantiate a specialization which has already been either implicitly or explicitly instantiated, this error will result.
W8077: Explicitly specializing an explicitly specialized class member makes no sense (C++) (see page 283)	Internal error. This warning is no longer generated by the compiler. The following code is illegal:
Informational messages (C++) (see page 284)	The compiler displays status information while compiling if you have checked "Show general messages" on the Compiler page of the Project Options dialog box. Most of the messages are self-explanatory and state information about compiling and linking; for example:
E2196: Cannot take address of member function 'function' (C++) (see page 284)	An expression takes the address of a class member function, but this member function was not found in the program being debugged. The evaluator issues this message.
F1002: Unable to create output file 'filename' (C++) (see page 284)	This error occurs if the work disk is full or write protected. This error also occurs if the output directory does not exist. *Solutions* If the disk is full, try deleting unneeded files and restarting the compilation. If the disk is write-protected, move the source files to a writeable disk and restart the compilation.
F1003: Error directive: 'message' (C++) (see page 284)	This message is issued when an `#error` directive is processed in the source file. 'message' is the text of the `#error` directive.
F1004: Internal compiler error (C++) (see page 284)	An error occurred in the internal logic of the compiler. This error shouldn't occur in practice, but is generated in the event that a more specific error message is not available.
F1006: Bad call of intrinsic function (C++) (see page 284)	You have used an intrinsic function without supplying a prototype. You may have supplied a prototype for an intrinsic function that was not what the compiler expected.
F1007: Irreducible expression tree (C++) (see page 285)	An expression on the indicated line of the source file caused the code generator to be unable to generate code. Avoid using the expression. Notify CodeGear if an expression consistently reproduces this error.
F1009: Unable to open input file 'filename' (C++) (see page 285)	This error occurs if the source file can't be found. Check the spelling of the name. Make sure the file is on the specified disk or directory. Verify that the proper directory paths are listed. If multiple paths are required, use a semicolon to separate them.

F1011: Register allocation failure (C++) (see page 285)	*Possible Causes* An expression on the indicated line of the source file was so complicated that the code generator could not generate code for it. *Solutions* Simplify the expression. If this does not solve the problem, avoid the expression. Notify CodeGear if an expression can consistently reproduce this error.
F1012: Compiler stack overflow (C++) (see page 285)	The compiler's stack has overflowed. This can be caused by a number of things, among them deeply nested statements in a function body (for example, if/else) or expressions with a large number of operands. You must simplify your code if this message occurs. Adding more memory to your system will not help.
F1013: Error writing output file (C++) (see page 285)	A DOS error that prevents the C++ IDE from writing an .OBJ, .EXE, or temporary file. *Solutions* Make sure that the Output directory in the Directories dialog box is a valid directory. Check that there is enough free disk space.
F1000: Compiler table limit exceeded (C++) (see page 285)	One of the compiler's internal tables overflowed. This usually means that the module being compiled contains too many function bodies. This limitation will not be solved by making more memory available to the compiler. You need to simplify the file being compiled.
F1005: Include files nested too deep (C++) (see page 286)	This message flags (directly or indirectly) recursive `#include` directives.
F1008: Out of memory (C++) (see page 286)	The total working storage is exhausted. This error can occur in the following circumstances: • Not enough virtual memory is available for compiling a particular file. In this case, shut down any other concurrent applications. You may also try to reconfigure your machine for more available virtual memory, or break up the source file being compiled into smaller separate components. You can also compile the file on a system with more available RAM. • The compiler has encountered an exceedingly complex or long expression at the line indicated and has insufficient reserves to parse it. Break the expression down into separate statements.... more (see page 286)
F1010: Unable to open 'filename' (C++) (see page 286)	This error occurs if the specified file can't be opened. Make sure the file is on the specified disk or directory. Verify the proper paths are listed. If multiple paths are required, use a semicolon to separate them.
E2000: 286/287 instructions not enabled (C++) (see page 286)	Use the -2 command-line compiler option to enable 286/287 opcodes. Be aware that the resulting code cannot be run on 8086- and 8088-based machines.
Abnormal program termination (C++) (see page 286)	The program called abort because there wasn't enough memory to execute. This message can be caused by memory overwrites.
E2009: Attempt to grant or reduce access to 'identifier' (C++) (see page 286)	A C++ derived class can modify the access rights of a base class member, but only by restoring it to the rights in the base class. It can't add or reduce access rights.
E2011: Illegal to take address of bit field (C++) (see page 286)	It is not legal to take the address of a bit field, although you can take the address of other kinds of fields.
E2010: Cannot add or subtract relocatable symbols (C++) (see page 287)	The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant. Variables, procedures, functions, and labels are relocatable symbols.
E2013: 'function1' cannot be distinguished from 'function2' (C++) (see page 287)	The parameter type lists in the declarations of these two functions do not differ enough to tell them apart. Try changing the order of parameters or the type of a parameter in one declaration.

E2014: Member is ambiguous: 'member1' and 'member2' (C++) (see page 287)	You must qualify the member reference with the appropriate base class name. In C++ class 'class', member 'member' can be found in more than one base class, and it was not qualified to indicate which one you meant. This applies only in multiple inheritance, where the member name in each base class is not hidden by the same member name in a derived class on the same path. The C++ language rules require that this test for ambiguity be made before checking for access rights (private, protected, public). It is possible to get this message even though only one (or... more (see page 287)
E2015: Ambiguity between 'function1' and 'function2' (C++) (see page 287)	Both of the named overloaded functions could be used with the supplied parameters. This ambiguity is not allowed.
E2017: Ambiguous member name 'name' (C++) (see page 287)	Whenever a structure member name is used in inline assembly, such a name must be unique. (If it is defined in more than one structure, all of the definitions must agree as to its type and offset within the structures). In this case, an ambiguous member name has been used. For example:
E2019: 'identifier' cannot be declared in an anonymous union (C++) (see page 288)	The compiler found a declaration for a member function or static member in an anonymous union. Such unions can only contain data members.
E2020: Global anonymous union not static (C++) (see page 288)	In C++, a global anonymous union at the file level must be static.
E2022: Array size too large (C++) (see page 288)	The declared array is larger than 64K and the 'huge' keyword was not used. If you need an array of this size, either use the 'huge' modifier, like this:
E2024: Cannot modify a const object (C++) (see page 288)	This indicates an illegal operation on an object declared to be const, such as an assignment to the object.
E2025: Assignment to 'this' not allowed, use X::operator new instead (C++) (see page 288)	In early versions of C++, the only way to control allocation of class of objects was by assigning to the 'this' parameter inside a constructor. This practice is no longer allowed, because a better, safer, and more general technique is to define a member function operator new instead. For example:
E2026: Assembler statement too long (C++) (see page 288)	Inline assembly statements can't be longer than 480 bytes.
E2001: Constructors and destructors not allowed in __automated section (C++) (see page 289)	Only member function declarations are allowed in __automated sections.
E2002: Only __fastcall functions allowed in __automated section (C++) (see page 289)	The calling convention for functions declared in an __automated section must be __fastcall.
E2003: Data member definition not allowed in __automated section (C++) (see page 289)	Only member function declarations are allowed in __automated sections.
E2004: Only read or write clause allowed in property declaration in __automated section (C++) (see page 289)	Storage specifiers stored, default, and nodefault are not allowed in property declarations in __automated sections.
E2005: Redeclaration of property not allowed in __automated section (C++) (see page 290)	If you declare a property in an __automated section it has be a new declaration. Property hoisting is not allowed.
E2027: Must take address of a memory location (C++) (see page 290)	Your source file used the address-of operator (&) with an expression that can't be used that way; for example, a register variable.
E2028: operator -> must return a pointer or a class (C++) (see page 290)	The C++ operator -> function must be declared to either return a class or a pointer to a class (or struct or union). In either case, it must be something to which the -> operator can be applied.
E2029: 'identifier' must be a previously defined class or struct (C++) (see page 290)	You are attempting to declare 'identifier' to be a base class, but either it is not a class or it has not yet been fully defined. Correct the name or rearrange the declarations.
E2030: Misplaced break (C++) (see page 290)	The compiler encountered a break statement outside a switch or looping construct. You can only use break statements inside of switch statements or loops.
E2031: Cannot cast from 'type1' to 'type2' (C++) (see page 290)	A cast from type 'ident1' to type 'ident2' is not allowed. In C++, you cannot cast a member function pointer to a normal function pointer. For example:
E2033: Misplaced continue (C++) (see page 291)	The compiler encountered a continue statement outside a looping construct.
E2034: Cannot convert 'type1' to 'type2' (C++) (see page 291)	An assignment, initialization, or expression requires the specified type conversion to be performed, but the conversion is not legal. In C++, the compiler will convert one function pointer to another only if the signature for the functions are the same. Signature refers to the arguments and return type of the function. For example:
E2036: Conversion operator cannot have a return type specification (C++) (see page 292)	This C++ type conversion member function specifies a return type different from the type itself. A declaration for conversion function operator can't specify any return type.

E2037: The constructor 'constructor' is not allowed (C++) (see page 292)	Constructors of the form
E2039: Misplaced decimal point (C++) (see page 292)	The compiler encountered a decimal point in a floating-point constant as part of the exponent.
E2041: Incorrect use of default (C++) (see page 292)	The compiler found no colon after the default keyword.
E2042: Declare operator delete (void*) or (void*, size_t) (C++) (see page 293)	*E2043 Declare operator delete[] (void*) or (void*, size_t)*Compiler error Declare the operator delete with one of the following: 1.A single void* parameter, or 2.A second parameter of type size_t If you use the second version, it will be used in preference to the first version. The global operator delete can only be declared using the single-parameter form.
E2044: operator delete must return void (C++) (see page 293)	*E2044 operator delete[] must return void*Compiler error This C++ overloaded operator delete was declared in some other way. Declare the operator delete with one of the following: 1.A single void* parameter, or 2.A second parameter of type size_t If you use the second version, it will be used in preference to the first version. The global operator delete can only be declared using the single-parameter form.
E2045: Destructor name must match the class name (C++) (see page 293)	In a C++ class, the tilde (~) introduces a declaration for the class destructor. The name of the destructor must be same as the class name. In your source file, the ~ preceded some other name.
E2048: Unknown preprocessor directive: 'identifier' (C++) (see page 293)	The compiler encountered a # character at the beginning of a line. The directive name that followed the # was not one of the following: • define (see page 690) • else (see page 688) • endif • if • ifdef (see page 688) • ifndef (see page 688) • include (see page 692) • line (see page 693) • pragma • undef (see page 692)
E2046: Bad file name format in include directive OR Bad file name format in line directive (C++) (see page 294)	Include and line directive file names must be surrounded by quotes ("filename.h") or angle brackets (<filename.h>). The file name was missing the opening quote or angle bracket. If a macro was used, the resulting expansion text is not surrounded by quote marks.
E2051: Invalid use of dot (C++) (see page 294)	An identifier must immediately follow a period operator (.). This is a rare message that can only occur in some specialized inline assembly statements. *Example*
E2053: Misplaced elif directive (C++) (see page 294)	The compiler encountered an `#elif` directive without any matching `#if`, `#ifdef`, or `#ifndef` directive.
E2054: Misplaced else (C++) (see page 294)	The compiler encountered an `else` statement without a matching if statement. *Possible Causes* • An extra "else" statement • An extra semicolon • Missing braces • Some syntax error in a previous "if" statement
E2055: Misplaced else directive (C++) (see page 294)	The compiler encountered an `#else` directive without any matching `#if`, `#ifdef`, or `#ifndef` directive.
E2056: Misplaced endif directive (C++) (see page 295)	The compiler encountered an `#endif` directive without any matching `#if`, `#ifdef`, or `#ifndef` directive.
E2059: Unknown language, must be C or C++ (C++) (see page 295)	In the C++ construction

E2060: Illegal use of floating point (C++) (see page 295)	Floating-point operands are not allowed in these operators • shift (SHL, SHR) • bitwise Boolean (AND, OR, XOR, NOT) • conditional (? :) • indirection (*) • certain others The compiler found a floating-point operand with one of these prohibited operators.
E2061: Friends must be functions or classes (C++) (see page 295)	A friend of a C++ class must be a function or another class.
E2062: Invalid indirection (C++) (see page 295)	The indirection operator (*) requires a pointer as the operand. *Example*
E2063: Illegal initialization (C++) (see page 296)	Initializations must be one of the following: • constant expressions • the address of a global extern or static variable plus or minus a constant
E2064: Cannot initialize 'type1' with 'type2' (C++) (see page 296)	You are attempting to initialize an object of type 'type1' with a value of type 'type2' which is not allowed. The rules for initialization are essentially the same as for assignment.
E2068: 'identifier' is not a non-static data member and can't be initialized here (C++) (see page 296)	Only data members can be initialized in the initializers of a constructor. This message means that the list includes a static member or function member. Static members must be initialized outside of the class, for example:
E2069: Illegal use of member pointer (C++) (see page 296)	Pointers to class members can only be passed as arguments to functions, or used with the following operators: • assignment operators • comparison operators • `.*` • `–>*` • `?:` conditional (ternary) operator • `&&` logical AND (see page 531) operator • `||` logical OR (see page 559) operator The compiler has encountered a member pointer being used with a different operator. In order to call a member function pointer, one must supply an instance of the class for it to call upon. For example:
E2071: operator new must have an initial parameter of type size_t (C++) (see page 297)	*E2071 Operator new[] must have an initial parameter of type size_t*Compiler error Operator new can be declared with an arbitrary number of parameters. It must always have at least one, the amount of space to allocate.
E2072: Operator new[] must return an object of type void (C++) (see page 297)	This C++ overloaded operator new was declared in some other way.
E2075: Incorrect 'type' option: option (C++) (see page 297)	An error has occurred in either the configuration file or a command-line option. The compiler may not have recognized the configuration file parameter as legal; check for a preceding hyphen (-), or the compiler may not have recognized the command-line parameter as legal. This error can also occur if you use a `#pragma` option in your code with an invalid option.

E2076: Overloadable operator expected (C++) (see page 297)	Almost all C++ operators can be overloaded. These are the only ones that can't be overloaded: • the field-selection dot (.) • dot-star (.*) • double colon (::) • conditional expression (?:) The preprocessor operators (# and ##) are not C or C++ language operators and thus can't be overloaded. Other non-operator punctuation, such as semicolon (;), can't be overloaded.
E2080: 'function' must be declared with one parameter (C++) (see page 298)	This C++ operator function was incorrectly declared with more than one parameter.
E2077: 'operator' must be declared with one or no parameters (C++) (see page 298)	When operator ++ or operator -- is declared as a member function, it must be declared to take either: • No parameters (for the prefix version of the operator), or • One parameter of type int (for the postfix version)
E2079: 'function' must be declared with no parameters (C++) (see page 298)	This C++ operator function was incorrectly declared with parameters.
E2078: 'operator' must be declared with one or two parameters (C++) (see page 298)	When operator ++ or operator -- is declared as a non-member function, it must be declared to take either: • one parameter (for the prefix version of the operator), or • two parameters (for the postfix version)
E2081: 'function' must be declared with two parameters (C++) (see page 298)	This C++ operator function was incorrectly declared with other than two parameters.
E2082: 'identifier' must be a member function or have a parameter of class type (C++) (see page 298)	Most C++ operator functions must have an implicit or explicit parameter of class type. This operator function was declared outside a class and does not have an explicit parameter of class type.
E2083: Last parameter of 'operator' must have type 'int' (C++) (see page 299)	When a postfix operator ++ or operator -- is overloaded, the last parameter must be declared with the type int.
E2084: Parameter names are used only with a function body (C++) (see page 299)	When declaring a function (not defining it with a function body), you must use either empty parentheses or a function prototype. A list of parameter names only is not allowed. *Example declarations*
E2085: Invalid pointer addition (C++) (see page 299)	Your source file attempted to add two pointers together.
E2086: Illegal pointer subtraction (C++) (see page 299)	This is caused by attempting to subtract a pointer from a non-pointer.
E2087: Illegal use of pointer (C++) (see page 299)	Pointers can only be used with these operators: • addition(+) • subtraction(-) • assignment(=) • comparison(==) • indirection(*) • arrow(->) Your source file used a pointer with some other operator. *Example*
E2088: Bad syntax for pure function definition (C++) (see page 300)	Pure virtual functions are specified by appending "= 0" to the declaration, like this:

E2089: Identifier 'identifier' cannot have a type qualifier (C++) (see page 300)	A C++ qualifier class::identifier can't be applied here. A qualifier is not allowed on the following: • typedef names • function declarations (except definitions at the file level) • on local variables or parameters of functions • on a class member--except to use its own class as a qualifier (redundant but legal).
E2090: Qualifier 'identifier' is not a class or namespace name (C++) (see page 300)	The C++ qualifier in the construction qual::identifier is not the name of a struct or class.
E2092: Storage class 'storage class' is not allowed here (C++) (see page 300)	The given storage class is not allowed here. Probably two storage classes were specified, and only one can be given.
E2096: Illegal structure operation (C++) (see page 300)	Structures can only be used with dot (.), address-of (&) or assignment (=) operators, or be passed to or from a function as parameters. The compiler encountered a structure being used with some other operator.
E2104: Invalid use of template keyword (C++) (see page 300)	You can only use a template class name without specifying its actual arguments inside a template definition. Using a template class name without specifying its actual arguments outside a template definition is illegal.
E2108: Improper use of typedef 'identifier' (C++) (see page 301)	Your source file used a typedef symbol where a variable should appear in an expression. Check for the declaration of the symbol and possible misspellings.
E2109: Not an allowed type (C++) (see page 301)	Your source file declared some sort of forbidden type; for example, a function returning a function or array.
E2110: Incompatible type conversion (C++) (see page 301)	The cast requested can't be done.
E2113: Virtual function 'function1' conflicts with base class 'base' (C++) (see page 301)	A virtual function has the same argument types as one in a base class, but differs in one or more of the following: • Return type • Calling convention • Exception specification (throw list)
E2114: Multiple base classes require explicit class names (C++) (see page 301)	In a C++ class constructor, if there is more than one immediate base class, each base class constructor call in the constructor header must include the base class name.
E2115: Bit field too large (C++) (see page 301)	This error occurs when you supply a bit field with more than 16 bits.
E2116: Bit fields must contain at least one bit (C++) (see page 301)	You can't declare a named bit field to have 0 (or less than 0) bits. You can declare an unnamed bit field to have 0 bits. This is a convention used to force alignment of the following bit field to a byte boundary (or to a word boundary.
W8005: Bit fields must be signed or unsigned int (C++) (see page 302)	In ANSI C, bit fields may only be signed or unsigned int (not char or long, for example).
E2119: User break (C++) (see page 302)	You typed a Ctrl+Break while compiling in the IDE. (This is not an error, just a confirmation.)
E2111: Type 'typename' may not be defined here (C++) (see page 302)	Class and enumeration types may not be defined in a function return type, a function argument type, a conversion operator type, or the type specified in a cast. You must define the given type before using it in one of these contexts. `Note:` This error message is often the result of a missing semicolon (`;`) for a class declaration. You might want to verify that all the class declarations preceding the line on which the error occurred end with a semicolon.
E2121: Function call missing) (C++) (see page 302)	The function call argument list had some sort of syntax error, such as a missing or mismatched right parenthesis.
E2123: Class 'class' may not contain pure functions (C++) (see page 302)	The class being declared cannot be abstract, and therefore it cannot contain any pure functions.
E2126: Case bypasses initialization of a local variable (C++) (see page 302)	In C++ it is illegal to bypass the initialization of a local variable. This error indicates a case label that can transfer control past this local variable.
E2127: Case statement missing : (C++) (see page 302)	A case statement must have a constant expression followed by a colon. The expression in the case statement either was missing a colon or had an extra symbol before the colon.

E2128: Case outside of switch (C++) (see page 302)	The compiler encountered a case statement outside a switch statement. This is often caused by mismatched braces.
E2129: Character constant too long (or empty) (C++) (see page 303)	Character constants can only be one or two characters long.
E2133: Unable to execute command 'command' (C++) (see page 303)	The linker or assembler cannot be found, or possibly the disk is bad.
E2134: Compound statement missing closing brace (C++) (see page 303)	The compiler reached the end of the source file and found no closing brace. This is most commonly caused by mismatched braces.
E2137: Destructor for 'class' required in conditional expression (C++) (see page 303)	If the compiler must create a temporary local variable in a conditional expression, it has no good place to call the destructor because the variable might or might not have been initialized. The temporary can be explicitly created, as with classname(val, val), or implicitly created by some other code. You should recast your code to eliminate this temporary value.
E2135: Constructor/Destructor cannot be declared 'const' or 'volatile' (C++) (see page 303)	A constructor or destructor has been declared as `const` or `volatile`. This is not allowed.
E2138: Conflicting type modifiers (C++) (see page 303)	This occurs when a declaration is given that includes more than one addressing modifier on a pointer or more than one language modifier for a function. Only one language modifier (for example, `__cdecl`, `__pascal`, or `__fastcall`) can be given for a function.
E2136: Constructor cannot have a return type specification (C++) (see page 303)	C++ constructors have an implicit return type used by the compiler, but you can't declare a return type or return a value.
E2038: Cannot declare or define 'identifier' here: wrong namespace (C++) (see page 304)	You tried to declare a template in an illegal place or a namespace member outside of its namespace.
E2154: Cannot define 'identifier' using a namespace alias (C++) (see page 304)	You cannot use a namespace alias to define a namespace member outside of its namespace.
E2421: Cannot use local type 'identifier' as template argument (C++) (see page 304)	A local type was used in an actual template type argument, which is illegal.
E2035: Conversions of class to itself or base class not allowed (C++) (see page 304)	You tried to define a conversion operator to the same class or a base class.
E2139: Declaration missing ; (C++) (see page 304)	Your source file contained a struct or union field declaration that was not followed by a semicolon. Check previous lines for a missing semicolon.
E2140: Declaration is not allowed here (C++) (see page 304)	Declarations can't be used as the control statement for while, for, do, if, or switch statements.
E2141: Declaration syntax error (C++) (see page 304)	Your source file contained a declaration that was missing a symbol or had an extra symbol added to it. Check for a missing semicolon or parenthesis on that line or on previous lines.
E2142: Base class 'class' contains dynamically dispatchable functions (C++) (see page 304)	This error occurs when a class containing a DDVT function attempts to inherit DDVT functions from multiple parent classes. Currently, dynamically dispatched virtual tables do not support the use of multiple inheritance.
E2143: Matching base class function 'function' has different dispatch number (C++) (see page 305)	If a DDVT function is declared in a derived class, the matching base class function must have the same dispatch number as the derived function.
E2144: Matching base class function 'function' is not dynamic (C++) (see page 305)	If a DDVT function is declared in a derived class, the matching base class function must also be dynamic.
E2145: Functions 'function1' and 'function2' both use the same dispatch number (C++) (see page 305)	This error indicates a dynamically dispatched virtual table (DDVT) problem.
E2146: Need an identifier to declare (C++) (see page 305)	In this context, an identifier was expected to complete the declaration. This might be a typedef with no name, or an extra semicolon at file level. In C++, it might be a class name improperly used as another kind of identifier.
E2147: 'identifier' cannot start a parameter declaration (C++) (see page 305)	An undefined 'identifier' was found at the start of an argument in a function declarator. Often the type name is misspelled or the type declaration is missing. This is usually caused by not including the appropriate header file.
E2150: Type mismatch in default argument value (C++) (see page 305)	The default parameter value given could not be converted to the type of the parameter. The message "Type mismatch in default argument value" is used when the parameter was not given a name. When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch. That other message is most often "Cannot convert 'type1' to 'type2'" but the mismatch could be due to another reason.
E2152: Default expression may not use local variables (C++) (see page 306)	A default argument expression is not allowed to use any local variables or other parameters.

E2153: Define directive needs an identifier (C++) (see page 306)	The first non-whitespace character after a `#define` must be an identifier. The compiler found some other character.
E2155: Too many default cases (C++) (see page 306)	The compiler encountered more than one default statement in a single switch.
E2156: Default outside of switch (C++) (see page 306)	The compiler encountered a default statement outside a switch statement. This is most commonly caused by mismatched braces.
E2158: Operand of 'delete' must be non-const pointer (C++) (see page 306)	It is illegal to delete a variable that is not a pointer. It is also illegal to delete a pointer to a constant. For example:
E2159: Trying to derive a far class from the huge base 'base' (C++) (see page 306)	This error is no longer generated by the compiler.
E2160: Trying to derive a far class from the near base 'base' (C++) (see page 306)	If a class is declared (or defaults to) near, all derived classes must also be near.
E2161: Trying to derive a huge class from the far base 'base' (C++) (see page 307)	This error is no longer generated by the compiler.
E2162: Trying to derive a huge class from the near base 'base' (C++) (see page 307)	This error is no longer generated by the compiler.
E2163: Trying to derive a near class from the far base 'base' (C++) (see page 307)	If a class is declared (or defaults to) far, all derived classes must also be far.
E2164: Trying to derive a near class from the huge base 'base' (C++) (see page 307)	This error is no longer generated by the compiler.
E2165: Destructor cannot have a return type specification (C++) (see page 307)	C++ destructors never return a value, and you can't declare a return type or return a value.
E2166: Destructor for 'class' is not accessible (C++) (see page 307)	The destructor for this C++ class is protected or private, and can't be accessed here to destroy the class. If a class destructor is private, the class can't be destroyed, and thus can never be used. This is probably an error. A protected destructor can be accessed only from derived classes. This is a useful way to ensure that no instance of a base class is ever created, but only classes derived from it.
E2167: 'function' was previously declared with the language 'language' (C++) (see page 307)	Only one language modifier (cdecl pascal) can be given for a function. This function has been declared with different language modifiers in two locations.
E2168: Division by zero (C++) (see page 308)	Your source file contains a divide or remainder in a constant expression with a zero divisor.
E2169: 'identifier' specifies multiple or duplicate access (C++) (see page 308)	A base class can be declared public or private, but not both. This access specifier can appear no more than once for a base class.
E2170: Base class 'class' is included more than once (C++) (see page 308)	A C++ class can be derived from any number of base classes, but can be directly derived from a given class only once.
E2171: Body has already been defined for function 'function' (C++) (see page 308)	A function with this name and type was previously supplied a function body. A function body can only be supplied once. One cause of this error is not declaring a default constructor which you implement. For example:
E2172: Duplicate case (C++) (see page 308)	Each case of a switch statement must have a unique constant expression value.
E2175: Too many storage classes in declaration (C++) (see page 308)	A declaration can never have more than one storage class, either Auto, Register, Static, or Extern.
E2176: Too many types in declaration (C++) (see page 309)	A declaration can never have more than one basic type. Examples of basic types are: • char • class • int • float • double • struct • union • enum • typedef name
E2179: virtual specified more than once (C++) (see page 309)	The C++ reserved word "virtual" can appear only once in one member function declaration.

E2007: Dispid only allowed in __automated sections (C++) (see page 309)	The definition of dispids is only permitted in __automated sections.
Divide error (C++) (see page 309)	You tried to divide an integer by zero, which is illegal.
E2182: Illegal parameter to __emit__ (C++) (see page 309)	There are some restrictions on inserting literal values directly into your code with the __emit__ function. For example, you cannot give a local variable as a parameter to __emit__.
E2183: File must contain at least one external declaration (C++) (see page 309)	This compilation unit was logically empty, containing no external declarations. ANSI C and C++ require that something be declared in the compilation unit.
E2184: Enum syntax error (C++) (see page 310)	An `enum` declaration did not contain a properly formed list of identifiers.
E2185: The value for 'identifier' is not within the range of 'type-name' (C++) (see page 310)	You have attempted to assign a value that is out of the range of the specified type.
E2186: Unexpected end of file in comment started on line 'number' (C++) (see page 310)	The source file ended in the middle of a comment. This is normally caused by a missing close of comment (*/).
E2187: Unexpected end of file in conditional started on line 'number' (C++) (see page 310)	The source file ended before the compiler (or MAKE) encountered `#endif`. The `#endif` either was missing or misspelled. Every `#if` statement needs a matching `#endif` statement.
E2188: Expression syntax (C++) (see page 310)	This is a catch-all error message when the compiler parses an expression and encounters a serious error. *Possible Causes* This is most commonly caused by one of the following: • two consecutive operators • mismatched or missing parentheses • a missing semicolon on the previous statement. *Solutions* If the line where the error occurred looks syntactically correct, look at the line directly above for errors. Try moving the line with the error to a different location in the file and recompiling. If the error still occurs at the moved statement, the syntax error is occurring somewhere in that statement. If the error occurred... more (see page 310)
E2190: Unexpected closing brace (C++) (see page 311)	An extra right brace was encountered where none was expected. Check for a missing closing brace. *Useful Tip:* The IDE has a mechanism for finding a matching curly brace. If you put the cursor on the '{' or '}' character, hold down Ctrl, hit 'Q' and then '{' or '}', it will position the cursor on the matching brace.
E2189: extern variable cannot be initialized (C++) (see page 311)	The storage class extern applied to a variable means that the variable is being declared but not defined here--no storage is being allocated for it. Therefore, you can't initialize the variable as part of the declaration.
E2344: Earlier declaration of 'identifier' (C++) (see page 311)	This error message only shows up after the messages "Multiple declaration for 'identifier'" and "Type mismatch in redeclaration of 'identifier'". It tells you where the previous definition of the identifier in question was found by the compiler, so you don't have to search for it.
E2192: Too few parameters in call (C++) (see page 311)	This error message occurs when a call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given. Make certain that your call to a function has the same parameters as the function prototype.
E2193: Too few parameters in call to 'function' (C++) (see page 311)	A call to the named function (declared using a prototype) has too few arguments. Make certain that the parameters in the call to the function match the parameters of the function prototype.
E2194: Could not find file 'filename' (C++) (see page 311)	The compiler is unable to find the file supplied on the command line.
E2197: File name too long (C++) (see page 311)	The file name given in an `#include` directive was too long for the compiler to process. File names in DOS must be no more than 79 characters long.

E2195: Cannot evaluate function call (C++) (see page 312)	The error message is issued if someone tries to explicitly construct an object or call a virtual function. In integrated debugger expression evaluation, calls to certain functions (including implicit conversion functions, constructors, destructors, overloaded operators, and inline functions) are not supported.
E2198: Not a valid expression format type (C++) (see page 312)	Invalid format specifier following expression in the debug evaluate or watch window. A valid format specifier is an optional repeat value followed by a format character (c, d, f[n], h, x, m, p, r, or s).
E2200: Functions may not be part of a struct or union (C++) (see page 312)	This C struct or union field was declared to be of type function rather than pointer to function. Functions as fields are allowed only in C++.
Floating point error: Divide by 0 OR Floating point error: Domain OR Floating point error: Overflow (C++) (see page 312)	These fatal errors result from a floating-point operation for which the result is not finite: • Divide by 0 means the result is +INF or -INF exactly, such as 1.0/0.0. • Domain means the result is NAN (not a number), like 0.0/0.0. • Overflow means the result is +INF (infinity) or -INF with complete loss of precision, such as assigning 1e200*1e200 to a double.
Floating point error: Stack fault (C++) (see page 312)	The floating-point stack has been overrun. This error may be due to assembly code using too many registers or due to a misdeclaration of a floating-point function. The program prints the error message and calls abort and _exit. These floating-point errors can be avoided by masking the exception so that it doesn't occur, or by catching the exception with signal.
Floating point error: Partial loss of precision OR Floating point error: Underflow (C++) (see page 312)	These exceptions are masked by default, because underflows are converted to zero and losses of precision are ignored.
E2201: Too much global data defined in file (C++) (see page 313)	The sum of the global data declarations exceeds 64K bytes. This includes any data stored in the DGROUP (all global variables, literal strings, and static locals). *Solutions* Check the declarations for any array that might be too large. You can also remove variables from the DGROUP. Here's how: • Declare the variables as automatic. This uses stack space. • Dynamically allocate memory from the heap using calloc, malloc, or farmalloc for the variables. This requires the use of pointers. Literal strings are also put in the DGROUP. Get the file farstr.zip from our BBS to extract literal strings into their own segment.... more (see page 313)
E2203: Goto bypasses initialization of a local variable (C++) (see page 313)	In C++, it is illegal to bypass the initialization of a local variable. This error indicates a goto statement that can transfer control past this local variable.
E2204: Group overflowed maximum size: 'name' (C++) (see page 313)	The total size of the segments in a group (for example, DGROUP) exceeded 64K.
E2206: Illegal character 'character' (0x'value') (C++) (see page 313)	The compiler encountered some invalid character in the input file. The hexadecimal value of the offending character is printed. This can also be caused by extra parameters passed to a function macro.
E2207: Implicit conversion of 'type1' to 'type2' not allowed (C++) (see page 313)	When a member function of a class is called using a pointer to a derived class, the pointer value must be implicitly converted to point to the appropriate base class. In this case, such an implicit conversion is illegal.
E2208: Cannot access an inactive scope (C++) (see page 313)	You have tried to evaluate or inspect a variable local to a function that is currently not active. (This is an integrated debugger expression evaluation message.)

E2209: Unable to open include file 'filename' (C++) (see page 314)	The compiler could not find the named file. *Possible Causes* • The named file does not exist. • An `#include` file included itself. • You do not have FILES set in CONFIG.SYS on your root directory. *Solutions* • Verify that the named file exists. • Set FILES = 20 in CONFIG.SYS.
E2210: Reference member 'member' is not initialized (C++) (see page 314)	References must always be initialized, in the constructor for the class. A class member of reference type must have an initializer provided in all constructors for that class. This means you can't depend on the compiler to generate constructors for such a class, because it has no way of knowing how to initialize the references.
E2212: Function defined inline after use as extern (C++) (see page 314)	Functions can't become inline after they have already been used. Either move the inline definition forward in the file or delete it entirely. The compiler encountered something like:
E2211: Inline assembly not allowed in inline and template functions (C++) (see page 314)	The compiler can't handle inline assembly statements in a C++ inline or template function. You could eliminate the inline assembly code or, in the case of an inline function, make this a macro, and remove the inline storage class.
F1001: Internal code generator error (C++) (see page 315)	An error has occurred in the internal logic of the code generator. Contact CodeGear technical support.
E2413: Invalid template declaration (C++) (see page 315)	After the declarator of a template member, either a semicolon, an initialization, or a body was expected, but some other, illegal token was found. This message appears when a template member is declared outside of the template, but the syntax was wrong.
E2070: Invalid use of namespace 'identifier' (C++) (see page 315)	A namespace identifier was used in an illegal way, for example, in an expression.
E2214: Cannot have a 'non-inline function/static data' in a local class (C++) (see page 315)	All members of classes declared local to a function must be entirely defined in the class definition. This means that local classes cannot contain any static data members, and all of their member functions must have bodies defined within the class definition.
E2215: Linkage specification not allowed (C++) (see page 315)	Linkage specifications such as extern "C" are only allowed at the file level. Move this function declaration out to the file level.
E2216: Unable to create turboc.$ln (C++) (see page 315)	The compiler cannot create the temporary file TURBOC.$LN because it cannot access the disk or the disk is full.
E2218: Templates can only be declared at namespace or class scope (C++) (see page 315)	Templates cannot be declared inside classes or functions. They are only allowed in the global scope, or file level. For example:
E2217: Local data exceeds segment size limit (C++) (see page 316)	The local variables in the current function take up more than 64K.
E2219: Wrong number of arguments in call of macro 'macro' (C++) (see page 316)	Your source file called the named macro with an incorrect number of arguments.
E2220: Invalid macro argument separator (C++) (see page 316)	In a macro definition, arguments must be separated by commas. The compiler encountered some other character after an argument name. This is correct:
E2221: Macro argument syntax error (C++) (see page 316)	An argument in a macro definition must be an identifier. The compiler encountered some non-identifier character where an argument was expected.
E2222: Macro expansion too long (C++) (see page 316)	A macro can't expand to more than 4,096 characters.
E2223: Too many decimal points (C++) (see page 316)	The compiler encountered a floating-point constant with more than one decimal point.
E2224: Too many exponents (C++) (see page 316)	The compiler encountered more than one exponent in a floating-point constant.
E2225: Too many initializers (C++) (see page 316)	The compiler encountered more initializers than were allowed by the declaration being initialized.
E2226: Extra parameter in call (C++) (see page 317)	A call to a function, via a pointer defined with a prototype, had too many arguments.
E2227: Extra parameter in call to function (C++) (see page 317)	A call to the named function (which was defined with a prototype) had too many arguments given in the call.
E2228: Too many error or warning messages (C++) (see page 317)	There were more errors or warnings than allowed.

E2233: Cannot initialize a class member here (C++) (see page 317)	Individual members of structs, unions, and C++ classes can't have initializers. A struct or union can be initialized as a whole using initializers inside braces. A C++ class can only be initialized by the use of a constructor.
E2232: Constant/Reference member 'member' in class without constructors (C++) (see page 317)	A class that contains constant or reference members (or both) must have at least one user-defined constructor. Otherwise, there would be no way to ever initialize such members.
E2229: Member 'member' has the same name as its class (C++) (see page 317)	A static data member, enumerator, member of an anonymous union, or nested type cannot have the same name as its class. Only a member function or a non-static member can have a name that is identical to its class.
E2234: Memory reference expected (C++) (see page 317)	The built-in assembler requires a memory reference. You probably forgot to put square brackets around an index register operand.
E2231: Member 'member' cannot be used without an object (C++) (see page 318)	This means that you have written class::member where 'member' is an ordinary (non-static) member, and there is no class to associate with that member. For example, it is legal to write this:
E2235: Member function must be called or its address taken (C++) (see page 318)	A reference to a member function must be called, or its address must be taken with & operator. In this case, a member function has been used in an illegal context. For example:
O2237: DPMI programs must use the large memory model (C++) (see page 318)	The compiler no longer issues this error.
E2238: Multiple declaration for 'identifier' (C++) (see page 318)	This identifier was improperly declared more than once. This might be caused by conflicting declarations such as: • int a; double a; • a function declared two different ways, or • a label repeated in the same function, or • some declaration repeated other than an extern function or a simple variable This can also happen by inadvertently including the same header file twice. For example, given:
E2239: 'identifier' must be a member function (C++) (see page 319)	Most C++ operator functions can be members of classes or ordinary non-member functions, but these are required to be members of classes: • operator = • operator -> • operator ( ) • type conversions This operator function is not a member function but should be.
E2240: Conversion of near pointer not allowed (C++) (see page 319)	A near pointer cannot be converted to a far pointer in the expression evaluation box when a program is not currently running. This is because the conversion needs the current value of DS in the user program, which doesn't exist.
E2243: Array allocated using 'new' may not have an initializer (C++) (see page 319)	When initializing a vector (array) of classes, you must use the constructor that has no arguments. This is called the default constructor, which means that you can't supply constructor arguments when initializing such a vector.
E2244: 'new' and 'delete' not supported (C++) (see page 319)	The integrated debugger does not support the evaluation of the new and delete operators.
E2245: Cannot allocate a reference (C++) (see page 319)	You have attempted to create a reference using the new operator. This is illegal, because references are not objects and can't be created through new.
E2309: Inline assembly not allowed (C++) (see page 320)	Your source file contains inline assembly language statements and you are compiling it from within the integrated environment. You must use the BCC command to compile this source file from the DOS command line.
E2250: No base class to initialize (C++) (see page 320)	This C++ class constructor is trying to implicitly call a base class constructor, but this class was declared with no base classes. Check your declarations.

E2254: : expected after private/protected/private (C++) (see page 320)	When used to begin a private, protected, or public section of a C++ class, the reserved words "private," "protected," and "public" must be followed by a colon.
E2255: Use :: to take the address of a member function (C++) (see page 320)	If f is a member function of class c, you take its address with the syntax
E2256: No : following the ? (C++) (see page 320)	The question mark (?) and colon (:) operators do not match in this expression. The colon might have been omitted, or parentheses might be improperly nested or missing.
E2257: , expected (C++) (see page 320)	A comma was expected in a list of declarations, initializations, or parameters. This problem is often caused by a missing syntax element earlier in the file or one of its included headers.
E2258: Declaration was expected (C++) (see page 320)	A declaration was expected here but not found. This is usually caused by a missing delimiter such as a comma, semicolon, right parenthesis, or right brace.
E2259: Default value missing (C++) (see page 321)	When a C++ function declares a parameter with a default value, all of the following parameters must also have default values. In this declaration, a parameter with a default value was followed by a parameter without a default value.
E2260: Default value missing following parameter 'parameter' (C++) (see page 321)	All parameters following the first parameter with a default value must also have defaults specified.
E2263: Exception handling not enabled (C++) (see page 321)	A 'try' block was found with the exception handling disabled.
E2264: Expression expected (C++) (see page 321)	An expression was expected here, but the current symbol can't begin an expression. This message might occur where the controlling expression of an if or while clause is expected or where a variable is being initialized. This message is often due to a symbol that is missing or has been added.
E2266: No file names given (C++) (see page 321)	The command line contained no file names. You must specify a source file name.
E2265: No file name ending (C++) (see page 321)	The file name in an `#include` statement was missing the correct closing quote or angle bracket.
E2271: Goto statement missing label (C++) (see page 321)	The `goto` keyword must be followed by an identifier.
E2272: Identifier expected (C++) (see page 321)	An identifier was expected here, but not found. In C, an identifier is expected in the following situations: • in a list of parameters in an old-style function header • after the reserved words struct or union when the braces are not present, and • as the name of a member in a structure or union (except for bit fields of width 0). In C++, an identifier is also expected in these situations: • in a list of base classes from which another class is derived, following a double colon (::), and • after the reserved word "operator" when no operator symbol is present.
E2275: Opening brace expected (C++) (see page 322)	A left brace was expected at the start of a block or initialization.
E2276: (expected (C++) (see page 322)	A left parenthesis was expected before a parameter list.
E2274: < expected (C++) (see page 322)	The keyword `template` was not followed by <. Every template declaration must include the template formal parameters enclosed within < >, immediately following the template keyword.
E2277: Lvalue required (C++) (see page 322)	The left side of an assignment operator must be an addressable expression. Addressable expressions include the following: • numeric or pointer variables • structure field references or indirection through a pointer • a subscripted array element
E2278: Multiple base classes not supported for Delphi classes (C++) (see page 322)	Delphi style classes cannot have multiple base classes.
E2280: Member identifier expected (C++) (see page 322)	The name of a structure or C++ class member was expected here, but not found. The right side of a dot (.) or arrow (->) operator must be the name of a member in the structure or class on the left of the operator.

E2279: Cannot find default constructor to initialize member 'identifier' (C++) (see page 323)	When the following occurs 1.A C++ class 'class1' contains a member of class 'class2,' and 2.You want to construct an object of type 'class1' (but not from another object of type 'class1'). There must be a constructor class2::class2() so that the member can be constructed. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class 'class2'. In that case, the compiler will not supply the default constructor automatically (you must supply one.
E2310: Only member functions may be 'const' or 'volatile' (C++) (see page 323)	Something other than a class member function has been declared `const` or `volatile`.
E2311: Non-virtual function 'function' declared pure (C++) (see page 323)	Only virtual functions can be declared pure, because derived classes must be able to override them.
E2283: Use . or -> to call 'function' (C++) (see page 323)	You attempted to call a member function without providing an object. This is required to call a member function.
E2284: Use . or -> to call 'member', or & to take its address (C++) (see page 323)	A reference to a non-static class member without an object was encountered. Such a member can't be used without an object, or its address must be taken with the & operator.
E2285: Could not find a match for 'argument(s)' (C++) (see page 323)	No C++ function could be found with parameters matching the supplied arguments. Check parameters passed to function or overload function for parameters that are being passed.
E2286: Overloaded function resolution not supported (C++) (see page 324)	In integrated debugger expression evaluation, resolution of overloaded functions or operators is not supported, not even to take an address.
E2287: Parameter 'number' missing name (C++) (see page 324)	In a function definition header, this parameter consisted only of a type specifier 'number' with no parameter name. This is not legal in C. (It is allowed in C++, but there's no way to refer to the parameter in the function.)
E2288: Pointer to structure required on left side of -> or ->* (C++) (see page 324)	Nothing but a pointer is allowed on the left side of the arrow (->) in C or C++. In C++ a -> operator is allowed.
E2290: 'code' missing] (C++) (see page 324)	This error is generated if any of the following occur: • Your source file declared an array in which the array bounds were not terminated by a right bracket. • The array specifier in an operator is missing a right bracket. • The operator [ ] was declared as operator [. • A right bracket is missing from a subscripting expression. Add the bracket or fix the declaration. Check for a missing or extra operator or mismatched parentheses.
E2291: brace expected (C++) (see page 324)	A right brace was expected at the end of a block or initialization.
E2292: Function should return a value (C++) (see page 324)	Your source file declared the current function to return some type other than `int` or `void`, but the compiler encountered a return with no value. This usually indicates some sort of error. Functions declared as returning `int` are exempt because older versions of C did not support `void` function return types.
E2293:) expected (C++) (see page 325)	A right parenthesis was expected at the end of a parameter list.
E2294: Structure required on left side of . or .* (C++) (see page 325)	The left side of a dot (.) operator (or C++ dot-star operator, .*) must evaluate to a structure type. In this case it did not. This error can occur when you create an instance of a class using empty parentheses, and then try to access a member of that 'object'.
E2312: 'constructor' is not an unambiguous base class of 'class' (C++) (see page 325)	A C++ class constructor is trying to call a base class constructor 'constructor.' This error can also occur if you try to change the access rights of 'class::constructor.' Check your declarations.
E2313: Constant expression required (C++) (see page 325)	Arrays must be declared with constant size. This error is commonly caused by misspelling a `#define` constant.
E2296: Templates not supported (C++) (see page 325)	An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

E2314: Call of nonfunction (C++) (see page 325)	The name being called is not declared as a function. This is commonly caused by incorrectly declaring the function or misspelling the function name.
E2321: Declaration does not specify a tag or an identifier (C++) (see page 325)	This declaration doesn't declare anything. This may be a struct or union without a tag or a variable in the declaration. C++ requires that something be declared. For example:
E2297: 'this' can only be used within a member function (C++) (see page 326)	In C++, "this" is a reserved word that can be used only within class member functions.
E2316: 'identifier' is not a member of 'struct' (C++) (see page 326)	You are trying to reference 'identifier' as a member of 'struct', but it is not a member. Check your declarations.
E2317: 'identifier' is not a parameter (C++) (see page 326)	In the parameter declaration section of an old-style function definition, 'identifier' is declared but not listed as a parameter. Either remove the declaration or add 'identifier' as a parameter.
E2319: 'identifier' is not a public base class of 'classtype' (C++) (see page 326)	The right operand of a .*, ->*, or ::operator was not a pointer to a member of a class that is either identical to (or an unambiguous accessible base class of) the left operand's class type.
E2320: Expression of scalar type expected (C++) (see page 326)	The !, ++, and -- operators require an expression of scalar type. Only these types are allowed: • char • short • int • long • enum • float • double • long double • pointer
E2302: No type information (C++) (see page 327)	The integrated debugger has no type information for this variable. Ensure that you've compiled the module with debug information. If it has, the module may have been compiled by another compiler or assembler.
E2303: Type name expected (C++) (see page 327)	One of these errors has occurred: • In declaring a file-level variable or a struct field, neither a type name nor a storage class was given. • In declaring a typedef, no type for the name was supplied. • In declaring a destructor for a C++ class, the destructor name was not a type name (it must be the same name as its class). • In supplying a C++ base class name, the name was not the name of a class.
E2304: 'Constant/Reference' variable 'variable' must be initialized (C++) (see page 327)	This C++ object is declared constant or as a reference, but is not initialized. It must be initialized at the point of declaration.
E2305: Cannot find 'class::class' ('class'&) to copy a vector OR Cannot find 'class'::operator=('class'&) to copy a vector (C++) (see page 327)	When a C++ class 'class1' contains a vector (array) of class 'class2', and you want to construct an object of type 'class1' from another object of type 'class 1', you must use this constructor:
E2306: Virtual base classes not supported for Delphi classes (C++) (see page 328)	Delphi style classes cannot be derived virtually, not even from other Delphi style classes.
E2308: do statement must have while (C++) (see page 328)	Your source file contained a do statement that was missing the closing while keyword.
E2322: Incorrect number format (C++) (see page 328)	The compiler encountered a decimal point in a hexadecimal number.
E2324: Numeric constant too large (C++) (see page 328)	String and character escape sequences larger than hexadecimal or octal 77 can't be generated. Two-byte character constants can be specified by using a second backslash. For example,

E2282: Namespace name expected (C++) (see page 328)	The name of a namespace symbol was expected.
E2334: Namespace member 'identifier' declared outside its namespace (C++) (see page 328)	Namespace members must be declared inside their namespace. You can only use explicit qualification to define a namespace member (for example, to give a body for a function declared in a namespace). The declaration itself must be inside the namespace.
E2325: Illegal octal digit (C++) (see page 329)	The compiler found an octal constant containing a non-octal digit (8 or 9).
E2329: Invalid combination of opcode and operands (C++) (see page 329)	The built-in assembler does not accept this combination of operands. *Possible causes* • There are too many or too few operands for this assembler opcode. • The number of operands is correct, but their types or order do not match the opcode.
E2327: Operators may not have default argument values (C++) (see page 329)	It is illegal for overloaded operators to have default argument values.
E2330: Operator must be declared as function (C++) (see page 329)	An overloaded operator was declared with something other than function type. For example:
E2333: Class member 'member' declared outside its class (C++) (see page 329)	C++ class member functions can be declared only inside the class declaration. Unlike nonmember functions, they can't be declared multiple times or at other locations.
E2335: Overloaded 'function name' ambiguous in this context (C++) (see page 329)	The only time an overloaded function name can be used or assigned without actually calling the function is when a variable or parameter of the correct function pointer type is initialized or assigned the address of the overload function. In this case, an overloaded function name has been used in some other context, for example, the following code will generate this error:
E2339: Cannot overload 'main' (C++) (see page 330)	You cannot overload main.
E2336: Pointer to overloaded function 'function' doesn't match 'type' (C++) (see page 330)	A variable or parameter is assigned (or initialized with) the address of an overloaded function. However, the type of the variable or parameter doesn't match any of the overloaded functions with the specified name.
E2337: Only one of a set of overloaded functions can be "C" (C++) (see page 330)	C++ functions are by default overloaded, and the compiler assigns a new name to each function. If you wish to override the compiler's assigning a new name by declaring the function `extern` "C", you can do this for only one of a set of functions with the same name. (Otherwise the linker would find more than one global function with the same name.)
E2338: Overlays only supported in medium, large, and huge memory models (C++) (see page 330)	The compiler no longer issues this error.
E2340: Type mismatch in parameter 'number' (C++) (see page 330)	The function called, via a function pointer, was declared with a prototype. However, the given parameter number (counting left to right from 1) could not be converted to the declared parameter type. When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch. That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.
E2341: Type mismatch in parameter 'number' in call to 'function' (C++) (see page 331)	Your source file declared the named function with a prototype, and the given parameter number (counting left to right from 1) could not be converted to the declared parameter type. When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch. That other message is usually "Cannot convert 'type1' to 'type2'", but the mismatch might be due to many other reasons.
E2342: Type mismatch in parameter 'parameter' (C++) (see page 331)	Your source file declared the function called via a function pointer with a prototype. However, the named parameter could not be converted to the declared parameter type. When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch. That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

E2343: Type mismatch in parameter 'parameter' in call to 'function' (C++) (see page 331)	Your source file declared the named function with a prototype, and the named parameter could not be converted to the declared parameter type. When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch. That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.
E2345: Access can only be changed to public or protected (C++) (see page 331)	A C++ derived class can modify the access rights of a base class member, but only to public or protected. A base class member can't be made private.
E2349: Nonportable pointer conversion (C++) (see page 331)	An implicit conversion between a pointer and an integral type is required, but the types are not the same size. You must use an explicit cast. This conversion might not make any sense, so be sure this is what you want to do.
E2350: Cannot define a pointer or reference to a reference (C++) (see page 332)	It is illegal to have a pointer to a reference or a reference to a reference.
E2352: Cannot create instance of abstract class 'class' (C++) (see page 332)	Abstract classes (those with pure virtual functions) can't be used directly, only derived from. When you derive an abstract base class, with the intention to instantiate instances of this derived class, you must override each of the pure virtual functions of the base class exactly as they are declared. For example:
E2354: Two operands must evaluate to the same type (C++) (see page 332)	The types of the expressions on both sides of the colon in the conditional expression operator (?:) must be the same, except for the usual conversions. These are some examples of usual conversions • `char` to `int` • `float` to `double` • void* to a particular pointer In this expression, the two sides evaluate to different types that are not automatically converted. This might be an error or you might merely need to cast one side to the type of the other. When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.... more (see page 332)
E2355: Recursive template function: 'x' instantiated 'y' (C++) (see page 333)	The compiler has detected a recursive template function instance. For example:
E2356: Type mismatch in redeclaration of 'identifier' (C++) (see page 333)	Your source file redeclared a variable with a different type than was originally declared for the variable. *Possible Causes* This can occur if a function is called and subsequently declared to return something other than an integer. *Solutions* If this has happened, you must declare the function before the first call to it.
E2357: Reference initialized with 'type1', needs lvalue of type 'type2' (C++) (see page 333)	A reference variable that is not declared constant must be initialized with an lvalue of the appropriate type. In this case, the initializer either wasn't an lvalue, or its type didn't match the reference being initialized.
E2358: Reference member 'member' needs a temporary for initialization (C++) (see page 333)	You provided an initial value for a reference type that was not an lvalue of the referenced type. This requires the compiler to create a temporary for the initialization. Because there is no obvious place to store this temporary, the initialization is illegal.

E2360: Invalid register combination (e.g. [BP+BX]) (C++) (see page 334)	The built-in assembler detected an illegal combination of registers in an instruction. These are valid index register combinations: • [BX] • [BP] • [SI] • [DI] • [BX+SI] • [BX+DI] • [BP+SI] • [BP+DI] Other index register combinations are not allowed.
E2361: 'specifier' has already been included (C++) (see page 334)	This type specifier occurs more than once in this declaration. Delete or change one of the occurrences.
E2362: Repeat count needs an lvalue (C++) (see page 334)	The expression before the comma (,) in the Watch or Evaluate window must be an accessible region of storage. For example, expressions like this one are not valid:
E2363: Attempting to return a reference to local variable 'identifier' (C++) (see page 334)	This C++ function returns a reference type, and you are trying to return a reference to a local (auto) variable. This is illegal, because the variable referred to disappears when the function exits. You can return a reference to any static or global variable, or you can change the function to return a value instead.
E2364: Attempting to return a reference to a local object (C++) (see page 334)	You attempted to return a reference to a temporary object in a function that returns a reference type. This may be the result of a constructor or a function call. This object will disappear when the function returns, making the reference illegal.
E2365: Member pointer required on right side of .* or ->* (C++) (see page 335)	The right side of a C++ dot-star (.*) or an arrow star (->*) operator must be declared as a pointer to a member of the class specified by the left side of the operator. In this case, the right side is not a member pointer.
E2366: Can't inherit non-RTTI class from RTTI base OR E2367 Can't inherit RTTI class from non-RTTI base (C++) (see page 335)	When virtual functions are present, the RTTI attribute of all base classes must match that of the derived class.
E2368: RTTI not available for expression evaluation (C++) (see page 335)	Expressions requiring RTTI are not supported by the expression evaluator in the integrated debugger. This error message is only issued by the expression evaluator (if you try to Inspect, Watch, or Evaluate), not by the compiler.
E2371: sizeof may not be applied to a bit field (C++) (see page 335)	sizeof returns the size of a data object in bytes, which does not apply to a bit field.
E2372: sizeof may not be applied to a function (C++) (see page 335)	sizeof can be applied only to data objects, not functions. You can request the size of a pointer to a function.
E2373: Bit field cannot be static (C++) (see page 335)	Only ordinary C++ class data members can be declared static, not bit fields.
E2374: Function 'function' cannot be static (C++) (see page 335)	Only ordinary member functions and the operators new and delete can be declared static. Constructors, destructors and other operators must not be static.
Stack overflow (C++) (see page 335)	This error is reported when you compile a function with the Test Stack Overflow option on, but there is not enough stack space to allocate the function's local variables. This error can also be caused by the following: • infinite recursion, or • an assembly language procedure that does not maintain the stack project • a large array in a function
E2376: statement missing (C++) (see page 336)	In a do, for, if, switch, or while statement, the compiler found no left parenthesis after the while keyword or test expression.
E2377: statement missing) (C++) (see page 336)	In a do, for, if, switch, or while statement, the compiler found no right parenthesis after the while keyword or test expression.
E2378: do-while or for statement missing ; (C++) (see page 336)	In a do or for statement, the compiler found no semicolon after the right parenthesis.

E2379: Statement missing ; (C++) (see page 336)	The compiler encountered an expression statement without a semicolon following it.
E2380: Unterminated string or character constant (C++) (see page 336)	The compiler found no terminating quote after the beginning of a string or character constant.
E2381: Structure size too large (C++) (see page 336)	Your source file declared a structure larger than 64K.
E2382: Side effects are not allowed (C++) (see page 336)	Side effects such as assignments, ++, or -- are not allowed in the debugger watch window. A common error is to use x = y (not allowed) instead of x == y to test the equality of x and y.
E2383: Switch selection expression must be of integral type (C++) (see page 336)	The selection expression in parentheses in a `switch` statement must evaluate to an integral type (`char, short, int, long, enum`). You might be able to use an explicit cast to satisfy this requirement.
E2433: Specialization after first use of template (C++) (see page 337)	An ANSI C++ rule requires that a specialization for a function template be declared before its first use. This error message is only issued when the ANSI conformance option (-A) is active.
E2384: Cannot call near class member function with a pointer of type 'type' (C++) (see page 337)	Also E2385 Cannot call near class member function 'function' with a pointer of type 'type' Member functions of near classes can't be called via a member pointer. This also applies to calls using pointers to members. (Remember, classes are near by default in the tiny, small, and medium memory models.) Either change the pointer to be near, or declare the class as far.
E2390: Type mismatch in parameter 'number' in template class name 'template' (C++) (see page 337)	The actual template argument value supplied for the given parameter did not exactly match the formal template parameter type. When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch. That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.
E2391: Type mismatch in parameter 'parameter' in template class name 'template' (C++) (see page 337)	The actual template argument value supplied for the given parameter did not exactly match the formal template parameter type. When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch. That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.
E2394: Too few arguments passed to template 'template' (C++) (see page 337)	A template class name was missing actual values for some of its formal parameters.
E2395: Too many arguments passed to template 'template' (C++) (see page 338)	A template class name specified too many actual values for its formal parameters.
E2396: Template argument must be a constant expression (C++) (see page 338)	A non-type template class argument must be a constant expression of the appropriate type. This includes constant integral expressions and addresses of objects or functions with external linkage or members.
E2401: Invalid template argument list (C++) (see page 338)	This error indicates that an illegal template argument list was found. In a template declaration, the keyword `template` must be followed by a list of formal arguments enclosed within < and > delimiters.
E2400: Nontype template argument must be of scalar type (C++) (see page 338)	A nontype formal template argument must have scalar type; it can have an integral, enumeration, or pointer type.
E2415: Template functions may only have 'type-arguments' (C++) (see page 338)	A function template was declared with a non-type argument. This is not allowed with a template function, as there is no way to specify the value when calling it.
E2425: 'member' is not a valid template type member (C++) (see page 338)	A member of a template with some actual arguments that depend on the formal arguments of an enclosing template was found not to be a member of the specified template in a particular instance.
E2428: Templates must be classes or functions (C++) (see page 338)	The declaration in a template declaration must specify either a class type or a function.
E2432: 'template' qualifier must name a template class or function instance' (C++) (see page 339)	When defining a template class member, the actual arguments in the template class name used as the left operand for the :: operator must match the formal arguments of the template class.
E2442: Two consecutive dots (C++) (see page 339)	Because an ellipsis contains three dots (...), and a decimal point or member selection operator uses one dot (.), two consecutive dots cannot legally occur in a C program.
E2443: Base class 'class' is initialized more than once (C++) (see page 339)	In a C++ class constructor, the list of initializations following the constructor header includes base class 'class' more than once.
E2444: Member 'member' is initialized more than once (C++) (see page 339)	In a C++ class constructor, the list of initializations following the constructor header includes the same member name more than once.

E2445: Variable 'identifier' is initialized more than once (C++) (see page 339)	This variable has more than one initialization. It is legal to declare a file level variable more than once, but it can have only one initialization (even if two are the same).
E2446: Function definition cannot be a typedef'ed declaration (C++) (see page 339)	In ANSI C, a function body cannot be defined using a typedef with a function Type. Redefine the function body.
E2132: Templates and overloaded operators cannot have C linkage (C++) (see page 339)	You tried to use a linkage specification with a template or overloaded operator. The most common cause for this error message is having the declaration wrapped in an extern "C" linkage specification.
E2447: 'identifier' must be a previously defined enumeration tag (C++) (see page 340)	This declaration is attempting to reference 'ident' as the tag of an enum type, but it has not been so declared. Correct the name, or rearrange the declarations.
E2448: Undefined label 'identifier' (C++) (see page 340)	The named label has a `goto` in the function, but no label definition.
E2449: Size of 'identifier' is unknown or zero (C++) (see page 340)	This identifier was used in a context where its size was needed. A struct tag might only be declared (the struct not defined yet), or an extern array might be declared without a size. It's illegal then to have some references to such an item (like sizeof) or to dereference a pointer to this type. Rearrange your declaration so that the size of 'identifier' is available.
E2450: Undefined structure 'structure' (C++) (see page 340)	The named structure was used in the source file, probably on a pointer to a structure, but had no definition in the source file. This is probably caused by a misspelled structure name or a missing declaration.
E2451: Undefined symbol 'identifier' (C++) (see page 340)	The named identifier has no declaration. *Possible Causes* • Actual declaration of identifier has been commented out. • Misspelling, either at this point or at the declaration. • An error in the declaration of the identifier. • The header file in which the identifier is declared was not included using `#include` Tools to help track down the problem: GREP (see page 170)
E2453: Size of the type 'identifier' is unknown or zero (C++) (see page 340)	This type was used in a context where its size was needed. For example, a struct tag might only be declared (the struct not defined yet). It's illegal then to have some references to such an item (like sizeof) or to dereference a pointer to this type. Rearrange your declarations so that the size of this type is available.
E2452: Size of the type is unknown or zero (C++) (see page 341)	This error message indicates that an array of unspecified dimension nested within another structure is initialized and the `-A` (ANSI) switch is on. For example:
E2454: union cannot be a base type (C++) (see page 341)	A union can't be used as a base type for another class type.
E2455: union cannot have a base type (C++) (see page 341)	In general, a C++ class can be of union type, but such a class can't be derived from any other class.
E2456: Union member 'member' is of type class with 'constructor' (or destructor, or operator =) (C++) (see page 341)	A union can't contain members that are of type class with user-defined constructors, destructors, or operator =.
E2461: '%s' requires run-time initialization/finalization (C++) (see page 341)	This message is issued when a global variable that is declared as __thread (a Win32-only feature) or a static data member of a template class is initialized with a non-constant initial value. This message is also issued when a global variable that is declared as __thread (a Win32-only feature) or a static data member of a template class has the type class with constructor or destructor.
E2464: 'virtual' can only be used with member functions (C++) (see page 341)	A data member has been declared with the `virtual` specifier. Only member functions can be declared virtual. For example:
E2465: unions cannot have virtual member functions (C++) (see page 342)	A union can't have virtual functions as its members.
E2466: void & is not a valid type (C++) (see page 342)	A reference always refers to an object, but an object cannot have the type void. Thus, the type void is not allowed.
E2467: 'Void function' cannot return a value (C++) (see page 342)	A function with a return type void contains a return statement that returns a value; for example, an int. Default = displayed

E2468: Value of type void is not allowed (C++) (see page 342)	A value of type `void` is really not a value at all, so it can't appear in any context where an actual value is required. Such contexts include the following: • the right side of an assignment • an argument of a function • the controlling expression of an if, for, or while statement.
E2469: Cannot use tiny or huge memory model with Windows (C++) (see page 342)	The compiler no longer issues this error.
E2006: CodeGuarded programs must use the large memory model and be targeted for Windows (C++) (see page 342)	The compiler no longer issues this error.
E2269: The function 'function' is not available (C++) (see page 342)	You tried to call a function that is known to the evaluator, but which was not present in the program being debugged, for example, an inline function.
E2124: Invalid function call (C++) (see page 343)	A requested function call failed because the function is not available in the program, a parameter cannot be evaluated, and so on. The evaluator issues this message.
E2213: Invalid 'expression' in scope override (C++) (see page 343)	The evaluator issues this message when there is an error in a scope override in an expression you are watching or inspecting. You can specify a symbol table, a compilation unit, a source file name, etc. as the scope of the expression, and the message will appear whenever the compiler cannot access the symbol table, compilation unit, or whatever.
E2236: Missing 'identifier' in scope override (C++) (see page 343)	The syntax of a scope override is somehow incomplete. The evaluator issues this message.
Pure virtual function called (C++) (see page 343)	This is a runtime error. It is generated if the body of a pure virtual function was never generated and somehow the compiler tried to call it.
E2095: String literal not allowed in this context (C++) (see page 343)	This error message is issued by the evaluator when a string literal appears in a context other than a function call.
Unexpected termination during compilation [Module Seg#:offset] OR Unexpected termination during linking [Module Seg#:offset] (C++) (see page 343)	If either of these errors occur, it indicates a catastrophic failure of the CodeGear tools. You should contact CodeGear to report the problem and to find a potential work around for your specific case. By isolating the test case as well as possible, you will increase the chance for either CodeGear or yourself to find a work around for the problem. Commonly, compiler failures can be worked around by moving the source code that is currently being compiled. Simple cases might be switching the order of variable declarations, or functions within the source module. Moving the scope and storage of... more (see page 343)
E2012: Cannot take address of 'main' (C++) (see page 344)	In C++, it is illegal to take the address of the main function.
E2016: Ambiguous override of virtual base member 'base_function': 'derived_function' (C++) (see page 344)	A virtual function in a virtual base class was overridden with two or more different functions along different paths in the inheritance hierarchy. For example,
E2021: Array must have at least one element (C++) (see page 344)	ANSI C and C++ require that an array be defined to have at least one element (objects of zero size are not allowed). An old programming trick declares an array element of a structure to have zero size, then allocates the space actually needed with malloc. You can still use this trick, but you must declare the array element to have (at least) one element if you are compiling in strict ANSI mode. Declarations (as opposed to definitions) of arrays of unknown size are still allowed. *Example*
E2023: Array of references is not allowed (C++) (see page 344)	It is illegal to have an array of references, because pointers to references are not allowed and array names are coerced into pointers.
E2032: Illegal use of closure pointer (C++) (see page 344)	A closure pointer variable is used incorrectly. Closure variables have limited usage. For instance, you can assign a function to a closure variable, and execute that function through the closure variable, but you cannot use a closure variable like a pointer variable.
E2040: Declaration terminated incorrectly (C++) (see page 345)	A declaration has an extra or incorrect termination symbol, such as a semicolon placed after a function body. A C++ member function declared in a class with a semicolon between the header and the opening left brace also generates this error.
E2047: Bad 'directive' directive syntax (C++) (see page 345)	A macro definition starts or ends with the ## operator, or contains the # operator that is not followed by a macro argument name. An example of this might be:
E2049: Class type 'type' cannot be marked as __declspec(delphireturn) (C++) (see page 345)	Classes marked as delphireturn are special classes that the compiler needs to recognize by name. These classes are predefined in the headers. Some of the delphireturn classes are Variant, AnsiString, and Currency. You cannot mark user-defined classes as delphireturn.

E2052: Dynamic function 'function' conflicts with base class 'class' (C++) (see page 345)	Some of the modifiers of this dynamic function conflict with the definition of the same function in the base class. The two functions should have the same modifiers. The following modifiers (among others) can cause conflicts: • __export • __import • declspec(naked) • declspec(package) • __fastcall
E2057: Exception specification not allowed here (C++) (see page 345)	Function pointer type declarations are not allowed to contain exception specifications.
E2058: Exception handling variable may not be used here (C++) (see page 346)	An attempt has been made to use one of the exception handling values that are restricted to particular exception handling constructs, such as GetExceptionCode().
E2065: Using namespace symbol 'symbol' conflicts with intrinsic of the same name (C++) (see page 346)	If you define a function in a namespace, which has a name that might be replaced by a call to an intrinsic when -Oi is on, it is not permitted to have a "using" declaration which refers to that member. For example, calls to "strcmp" are replaced by the intrinsic "__strcmp__" when -Oi is on. This means that the declaration "using N::strcmp;" would become "using N::__strcmp__", since the token replacement happens before the compiler's parser ever sees the tokens. An error displays in this case, because the compiler doesn't know how to process "N::__strcmp__".
E2067: 'main' must have a return type of int (C++) (see page 346)	In C++, function main has special requirements, one of which is that it cannot be declared with any return type other than int.
E2073: Nothing allowed after pragma option pop (C++) (see page 346)	The #pragma option pop can only be followed by comments, blanks, or end of line.
E2091: Functions cannot return arrays or functions (C++) (see page 346)	A function was defined to return an array or a function. Check to see if either the intended return was a pointer to an array or function (and perhaps the * is missing) or if the function definition contained a request for an incorrect datatype.
E2093: Operator 'operator' not implemented in type 'type' for arguments of the same type (C++) (see page 346)	The operator you are calling is not defined in this class. When you have an expression: x + x, where x is of type class X, the operator + has to be defined in class X and be accessible.
E2094: Operator 'operator' not implemented in type 'type' for arguments of type 'type' (C++) (see page 346)	The operator you are calling is not defined in this class. When you have an expression: x + x, where x is of type class X, the operator + has to be defined in class X and be accessible.
E2097: Explicit instantiation only allowed at file or namespace scope (C++) (see page 347)	The explicit instantiation operator "template" can only be used within global or namespace scope. It cannot be used to qualify a local class or a class member, for example.
E2098: Explicit specialization declarator "template<>" now required (C++) (see page 347)	When specializing a function, such as providing the definition for "foo<int>", so that foo behaves specially which called for the "int" argument, now requires that the declaration begin with an explicit specialization operator.
E2099: Explicit specialization only allowed at file or namespace scope (C++) (see page 347)	The explicit specialization operator template<> can only be used within global or namespace scope. It cannot be used to qualify a local class or a class member, for example.
E2101: 'export' keyword must precede a template declaration (C++) (see page 347)	The 'export' keyword can only occur before the keyword "template" ina template declaration. It cannot be used anywhere else.
E2103: Explicit instantiation must be used with a template class or function (C++) (see page 347)	The explicit instantiation operator "template" can only be used to refer to templates. It cannot be used with non-templates.
E2106: Explicit specialization must be used with a template class or function (C++) (see page 347)	The explicit specialization operator template<> can only be used in front of a template class or function. Using it with a normal class means nothing, and hence generates an error.
E2112: Unknown unit directive: 'directive' (C++) (see page 347)	You cannot use this name as a unit directive. Instead use one of the following unit directives: weak, smart_init, or deny.
E2118: Bit fields must have integral type (C++) (see page 348)	In C++, bit fields must have an integral type. This includes enumerations.
E2120: Cannot call 'main' from within the program (C++) (see page 348)	C++ does not allow recursive calls of main( ).
E2125: Compiler could not generate copy constructor for class 'class' OR Compiler could not generate default constructor for class 'class' OR Compiler could not generate operator = for class 'class' (C++) (see page 348)	Sometimes the compiler is required to generate a member function for the user. Whenever such a member function can't be generated due to applicable language rules, the compiler issues one of these error messages.
E2130: Circular property definition (C++) (see page 348)	Indicates that a property definition relies directly or indirectly on itself.
E2131: Objects of type 'type' cannot be initialized with { } (C++) (see page 348)	Ordinary C structures can be initialized with a set of values inside braces. C++ classes can only be initialized with constructors if the class has constructors, private members, functions, or base classes that are virtual.

E2148: Default argument value redeclared for parameter 'parameter' (C++) (see page 348)	When a parameter of a C++ function is declared to have a default value, this value can't be changed, redeclared, or omitted in any other declaration for the same function.
E2149: Default argument value redeclared (C++) (see page 349)	When a parameter of a C++ function is declared to have a default value, this value can't be changed, redeclared, or omitted in any other declaration for the same function.
E2151: Type mismatch in default value for parameter 'parameter' (C++) (see page 349)	The default parameter value given could not be converted to the type of the parameter. The message "Type mismatch in default argument value" is used when the parameter was not given a name. When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch. That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.
E2157: Deleting an object requires exactly one conversion to pointer operator (C++) (see page 349)	If a person uses the 'delete' operator on an object (note: not a pointer to an object, but an object itself), the standard requires that object to define exactly one "conversion to pointer operator" which will yield the pointer that gets freed. For example:
E2173: Duplicate handler for 'type1', already had 'type2' (C++) (see page 349)	It is not legal to specify two handlers for the same type.
E2174: The name handler must be last (C++) (see page 349)	In a list of catch handlers, if the specified handler is present, it must be the last handler in the list (that is, it cannot be followed by any more catch handlers).
E2177: Redeclaration of #pragma package with different arguments (C++) (see page 350)	You can have multiple #pragma package statements in a source file as long as they have the same arguments. This error occurs if the compiler encounters more than one #pragma package with different arguments in each.
E2178: VIRDEF name conflict for 'function' (C++) (see page 350)	The compiler must truncate mangled names to a certain length because of a name length limit that is imposed by the linker. This truncation may (in very rare cases) cause two names to mangle to the same linker name. If these names happen to both be VIRDEF names, the compiler issues this error message. The simplest workaround for this problem is to change the name of 'function' so that the conflict is avoided.
E2180: Dispid number already used by identifier (C++) (see page 350)	Dispids must be unique and the compiler checks for this.
E2181: Cannot override a 'dynamic/virtual' with a 'dynamic/virtual' function (C++) (see page 350)	When you declare a function dynamic, you cannot override this function in a derived class with a virtual function of the same name and type. Similarly when the function is declared virtual, you cannot override it with a dynamic one in a derived class.
E2202: Goto into an exception handler is not allowed (C++) (see page 350)	It is not legal to jump into a try block, or an exception handler that is attached to a try block.
E2205: Illegal type type in __automated section (C++) (see page 350)	Only certain types are allowed in __automated sections.
E2242: Specifier requires Delphi style class type (C++) (see page 351)	The stored, default, and nodefault storage specifiers are only allowed within property declarations of Delphi style class types.
E2247: 'member' is not accessible (C++) (see page 351)	You are trying to reference C++ class member 'member,' but it is private or protected and can't be referenced from this function. This sometimes happens when you attempt to call one accessible overloaded member function (or constructor), but the arguments match an inaccessible function. The check for overload resolution is always made before checking for accessibility. If this is the problem, try an explicit cast of one or more parameters to select the desired accessible function. Virtual base class constructors must be accessible within the scope of the most derived class. This is because C++ always constructs virtual base classes... more (see page 351)
E2248: Cannot find default constructor to initialize array element of type 'class' (C++) (see page 351)	When declaring an array of a class that has constructors, you must either explicitly initialize every element of the array, or the class must have a default constructor. The compiler will define a default constructor for a class unless you have defined any constructors for the class.
E2251: Cannot find default constructor to initialize base class 'class' (C++) (see page 352)	Whenever a C++ derived class 'class2' is constructed, each base class 'class1' must first be constructed. If the constructor for 'class2' does not specify a constructor for 'class1' (as part of 'class2's' header), there must be a constructor class1::class1() for the base class. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class 'class1'. In that case, the compiler will not supply the default constructor automatically--you must supply one.
E2252: 'catch' expected (C++) (see page 352)	In a C++ program, a 'try' block must be followed by at least one 'catch' block.

E2253: Calling convention must be attributed to the function type, not the closure (C++) (see page 352)	The calling convention is in the wrong place in the closure declaration. For example,
E2261: Use of dispid with a property requires a getter or setter (C++) (see page 353)	This property needs either a getter or a setter.
E2262: '__except' or '__finally' expected following '__try' (C++) (see page 353)	In C, a '__try block' must be followed by a '__except' or '__finally' handler block.
E2270: > expected (C++) (see page 353)	A new-style cast (for example, dynamic_cast) was found with a missing closing ">".
E2273: 'main' cannot be declared as static or inline (C++) (see page 353)	You cannot make main static or inline. For example, you cannot use static int main() or inline int main().
E2281: Identifier1 requires definition of Identifier2 as a pointer type (C++) (see page 353)	To use Identifier1, there needs to be a definition for Identifier2, which is a type. Example where __classid is the first identifier and TClass, which can be found in clx.h, is the second one:
E2289: __published or __automated sections only supported for Delphi classes (C++) (see page 353)	The compiler needs to generate a special kind of vtable for classes containing __published and __automated sections. Therefore, these sections are only supported for Delphi style classes.
E2298: Cannot generate 'function' from template function 'template' (C++) (see page 354)	A call to a template function was found, but a matching template function cannot be generated from the function template.
E2301: Cannot use templates in closure arguments -- use a typedef (C++) (see page 354)	When declaring a closure type, the arguments passed to that closure must be of a simple type. Templates are not accepted. To pass a reference to an object of template type to a closure, you must declare a typedef, which counts as a simple type name.
E2307: Type 'type' is not a defined class with virtual functions (C++) (see page 354)	A dynamic_cast was used with a pointer to a class type that is either undefined, or doesn't have any virtual member functions.
E2315: 'Member' is not a member of 'class', because the type is not yet defined (C++) (see page 354)	The member is being referenced while the class has not been fully defined yet. This can happen if you forward declare class X, declare a pointer variable to X, and reference a member through that pointer; for example:
E2318: 'type' is not a polymorphic class type (C++) (see page 354)	This error is generated if the -RT compiler option (for runtime type information) is disabled and either dynamic_cast was used with a pointer to a class or you tried to delete a pointer to an object of a class that has a virtual destructor
E2323: Illegal number suffix (C++) (see page 355)	A numeric literal is followed by a suffix that is not recognized by the compiler.
E2326: Use __declspec(spec1[, spec2]) to combine multiple __declspecs (C++) (see page 355)	When you want to use several __declspec modifiers, the compiler will complain if you don't combine them into one __declspec. For example:
E2328: Classes with properties cannot be copied by value (C++) (see page 355)	This error occurs if you attempt to use the default assignment operator. For example, the following code generates this error given two labels on a form:
E2331: Number of allowable option contexts exceeded (C++) (see page 355)	You have interspersed too many source-code option changes (using #pragma option) between template declarations. For example:
E2332: Variable 'variable' has been optimized and is not available (C++) (see page 355)	You have tried to inspect, watch, or otherwise access a variable which the optimizer removed. This variable is never assigned a value and has no stack location.
E2476: Cannot overload 'function' (C++) (see page 356)	You cannot overload the specified function. This error is displayed if you tried to declare a function with the same name as another function, but the redeclaration is not legal. For example, if both functions have the 'extern "C"' linkage type, only one 'extern "C"' function can have a given name.
E2346: 'x' access specifier of property 'property' must be a member function (C++) (see page 356)	Only member functions or data members are allowed in access specifications of properties.
E2347: Parameter mismatch in access specifier 'specifier' of property 'property' (C++) (see page 356)	The parameters of the member function used to access a property don't match the expected parameters.
E2348: Storage specifier not allowed for array properties (C++) (see page 356)	Array properties cannot have a storage specification.
E2351: Static data members not allowed in __published or __automated sections (C++) (see page 357)	Only nonstatic data members and member functions are allowed in __published or __automated sections.
E2353: Class 'classname' is abstract because of 'member = 0' (C++) (see page 357)	This message is issued immediately after the "Cannot create instance of abstract class 'classname' error message and is intended to make it easier to figure out why a particular class is considered abstract by the compiler. For example, consider the following example of an illegal attempt to instantiate an abstract class:
E2359: Reference member 'member' initialized with a non-reference parameter (C++) (see page 357)	An attempt has been made to bind a reference member to a constructor parameter. Since the parameter will cease to exist the moment the constructor returns to its caller, this will not work correctly.
E2369: Cannot use the result of a property assignment as an rvalue' (C++) (see page 358)	The result of a property assignment is an lvalue. This implies for instance that chained assignments of properties is not allowed; for example, x = y = 5, where both x and y are properties. Certain embedded assignments of properties can also produce errors; for example, x != (y = z), where y is a property.

E2370: Simple type name expected (C++) (see page 358)	To ensure interoperability between Delphi and C++, there are restrictions on the type names mentioned in the parameter lists of published closure types. The parameter types have to be simple type names with optional const modifier and pointer or reference notation. So when declaring a closure type, the arguments passed to that closure must be of a simple type. For example, templates are not accepted. To pass a reference to an object of template type to a closure, you must declare a typedef, which counts as a simple type name.
E2398: Template function argument 'argument' not used in argument types (C++) (see page 358)	The given argument was not used in the argument list of the function. The argument list of a template function must use all of the template formal arguments; otherwise, there is no way to generate a template function instance based on actual argument types.
E2419: Error while instantiating template 'template' (C++) (see page 358)	An error occurred during the instantiation of a particular template. This message always follows some other error message that indicates what actually went wrong. This message is displayed to help track down which template instantiation introduced the problem.
E2424: Template class nesting too deep: 'class' (C++) (see page 358)	The compiler imposes a certain limit on the level of template class nesting. This limit is usually only exceeded through a recursive template class dependency. When this nesting limit is exceeded, the compiler issues this error message for all of the nested template classes. This usually makes it easy to spot the recursion. This error message is always followed by the fatal error "Out of memory".
E2457: Delphi style classes must be caught by reference (C++) (see page 359)	You can only catch a Delphi style object by pointer.
E2458: Delphi classes have to be derived from Delphi classes (C++) (see page 359)	You cannot derive a Delphi style class from a non-Delphi style class.
E2459: Delphi style classes must be constructed using operator new (C++) (see page 359)	Delphi style classes cannot be statically defined. They have to be constructed on the heap.
E2460: Delphi style classes require exception handling to be enabled (C++) (see page 360)	If you are using Delphi style classes in your program, you cannot turn off exception handling (compiler option `-x-`) when compiling your source code.
E2463: 'base' is an indirect virtual base class of 'class' (C++) (see page 360)	You can't create a pointer to a C++ member of a virtual base class. You have attempted to create such a pointer (either directly, or through a cast) and access an inaccessible member of one of your base classes.
Null pointer assignment (C++) (see page 360)	When a small or medium memory model program exits, a check is made to determine if the contents of the first few bytes within the program's data segment have changed. These bytes would never be altered by a working program. If they have been changed, this message is displayed to inform you that (most likely) a value was stored to an uninitialized pointer. The program might appear to work properly in all other respects; however, this is a serious bug which should be attended to immediately. Failure to correct an uninitialized pointer can lead to unpredictable behavior (including locking the... more (see page 360)
E2268: Call to undefined function 'function' (C++) (see page 360)	Your source file declared the current function to return some type other than `void` in C++ (or `int` in C), but the compiler encountered a return with no value. All `int` functions are exempt in C because in old versions of C, there was no `void` type to indicate functions that return nothing.
E2375: Assembler stack overflow (C++) (see page 360)	The assembler ran out of memory during compilation. Review the portion of code flagged by the error message to ensure that it uses memory correctly.
Initializing enumeration with type (C++) (see page 360)	You're trying to initialize an enum variable to a different type. For example,
<name> is not a valid identifier (C++) (see page 361)	The identifier name is invalid. Ensure that the first character is a letter or an underscore (_). The characters that follow must be letters, digits, or underscores, and there can not be any spaces in the identifier.
Example for "Temporary used ..." error messages (C++) (see page 361)	In this example, function f requires a reference to an int, and c is a char:
Application is running (C++) (see page 361)	The application you tried to run is already running. For Windows, make sure the message loop of the program has properly terminated.
Printf/Scanf floating-point formats not linked (C++) (see page 361)	Floating-point formats contain formatting information that is used to manipulate floating-point numbers in certain runtime library functions, such as scanf() and atof(). Typically, you should avoid linking the floating-point formats (which take up about 1K) unless they are required by your application. However, you must explicitly link the floating-point formats for programs that manipulate fields in a limited and specific way. Refer to the following list of potential causes (listed from most common to least common) to determine how to resolve this error: • CAUSE: Floating point set to None. You set the floating-point option to None when it should... more (see page 361)

W8000: Ambiguous operators need parentheses (C++) (see page 362)	(Command-line option to display warning: -wamb) This warning is displayed whenever two shift, relational, or bitwise-Boolean operators are used together without parentheses. Also, an addition or subtraction operator that appears without parentheses with a shift operator will produce this warning.
W8060: Possibly incorrect assignment (C++) (see page 362)	(Command-line option to suppress warning: -w-pia) This warning is generated when the compiler encounters an assignment operator as the main operator of a conditional expression (part of an if, while, or do-while statement). This is usually a typographical error for the equality operator. If you want to suppress this warning, enclose the assignment in parentheses and compare the whole thing to zero explicitly. For example, this code
W8002: Restarting compile using assembly (C++) (see page 362)	(Command-line option to suppress warning: -w-asc) The compiler encountered an asm with no accompanying or `#pragma` inline statement. The compile restarts using assembly language capabilities. Default = On
W8003: Unknown assembler instruction (C++) (see page 362)	(Command-line option to suppress warning: -w-asm) The compiler encountered an inline assembly statement with a disallowed opcode or an unknown token. Check the spelling of the opcode or token. `Note:` You will get a separate error message from the assembler if you entered illegal assembler source code. This warning is off by default.
W8052: Base initialization without a class name is now obsolete (C++) (see page 362)	(Command-line option to suppress warning: -w-obi) Early versions of C++ provided for initialization of a base class by following the constructor header with just the base class constructor parameter list. It is now recommended to include the base class name. This makes the code much clearer, and is required when you have multiple base classes. *Old way*
E2117: Bit fields must be signed or unsigned int (C++) (see page 363)	(Command-line option to display warning: -wbbf) In ANSI C, bit fields may not be of type signed char or unsigned char. When you're not compiling in strict ANSI mode, the compiler allows these constructs, but flags them with this warning.
W8064: Call to function with no prototype (C++) (see page 363)	(Command-line option to suppress warning: -w-pro) This message is given if the "Prototypes required" warning is enabled and you call a function without first giving a prototype for that function.
W8065: Call to function 'function' with no prototype (C++) (see page 363)	This message is given if the "Prototypes required" warning is enabled and you call function 'function' without first giving a prototype for that function.
W8009: Constant is long (C++) (see page 363)	(Command-line option to display warning: -wcln) The compiler encountered one of the following: • a decimal constant greater than 32,767 or • an octal, hexadecimal, or decimal constant greater than 65,535 without a letter l or L following it The constant is treated as a long.
W8008: Condition is always true OR W8008 Condition is always false (C++) (see page 363)	(Command-line option to suppress warning: -w-ccc) Whenever the compiler encounters a constant comparison that (due to the nature of the value being compared) is always true or false, it issues this warning and evaluates the condition at compile time. For example:
W8012: Comparing signed and unsigned values (C++) (see page 364)	(Command-line option to suppress warning: -w-csu) Since the ranges of signed and unsigned types are different the result of an ordered comparison of an unsigned and a signed value might have an unexpected result.

W8010: Continuation character \ found in // comment (C++) (see page 364)	(Command-line option to suppress warning: -w-com) This warning message is issued when a C++ // comment is continued onto the next line with backslash line continuation. The intention is to warn about cases where lines containing source code unintentionally become part of a comment because that comment happened to end in a backslash. If you get this warning, check carefully whether you intend the line after the // comment to be part of the comment. If you don't, either remove the backslash or put some other character after it. If you do, it's probably better coding style to start the... more (see page 364)
W8080: 'identifier' is declared but never used (C++) (see page 364)	(Command-line option to display warning: -wuse) The specified identifier was never used. This message can occur in the case of either local or static variables. It occurs when the source file declares the named local or static variable as part of the block just ending, but the variable was never used. In the case of local variables, this warning occurs when the compiler encounters the closing brace of the compound statement or function. In the case of static variables, this warning occurs when the compiler encounters the end of the source file. For example:
W8014: Declaration ignored (C++) (see page 365)	(Command-line option to suppress warning: -w-dig) An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility. Default = On
W8068: Constant out of range in comparison (C++) (see page 365)	(Command-line option to suppress warning: -w-rng) Your source file includes a comparison involving a constant sub-expression that was outside the range allowed by the other sub-expression's type. For example, comparing an unsigned quantity to -1 makes no sense. To get an unsigned constant greater than 32,767 (in decimal), you should either • cast the constant to unsigned--for example, (unsigned) 65535, or • append a letter u or U to the constant--for example, 65535u. Whenever this message is issued, the compiler still generates code to do the comparison. If this code ends up always giving the same result (such as comparing a char... more (see page 365)
W8016: Array size for 'delete' ignored (C++) (see page 365)	(Command-line option to suppress warning: -w-dsz) The C++ IDE issues this warning when you've specified the array size when deleting an array. With the new C++ specification, you don't need to make this specification. The compiler ignores this construct. This warning lets older code compile.
W8082: Division by zero (C++) (see page 365)	(Command-line option to suppress warning: -w-zdi) A divide or remainder expression had a literal zero as a divisor.
W8018: Assigning 'type' to 'enumeration' (C++) (see page 365)	(Command-line option to suppress warning: -w-eas) Assigning an integer value to an enum type. This is an error in C++, but is reduced to a warning to give existing programs a chance to work.
W8006: Initializing 'identifier' with 'identifier' (C++) (see page 366)	(Command-line option to suppress warning: `-w-bei`) You're trying to initialize an `enum` variable to a different type. For example, the following initialization will result in this warning, because 2 is of type int, not type enum count:
W8001: Superfluous & with function (C++) (see page 366)	(Command-line option to display warning: -wamp) An address-of operator (&) is not needed with function name; any such operators are discarded.
W8020: 'identifier' is declared as both external and static (C++) (see page 366)	(Command-line option to suppress warning: -w-ext) This identifier appeared in a declaration that implicitly or explicitly marked it as global or external, and also in a static declaration. The identifier is taken as static. You should review all declarations for this identifier.

W8007: Hexadecimal value contains more than three digits (C++) (see page 366)	(Command-line option to suppress warning = -w-big) Under older versions of C, a hexadecimal escape sequence could contain no more than three digits. The ANSI standard allows any number of digits to appear as long as the value fits in a byte. This warning results when you have a long hexadecimal escape sequence with many leading zero digits (such as \x00045). Older versions of C would interpret such a string differently.
W8024: Base class 'class1' is also a base class of 'class2' (C++) (see page 366)	(Command-line option to suppress warning: -w-ibc) A class inherits from the same base class both directly and indirectly. It is best to avoid this non-portable construct in your program code.
W8022: 'function1' hides virtual function 'function2' (C++) (see page 367)	(Command-line option to suppress warning: -w-hid) A virtual function in a base class is usually overridden by a declaration in a derived class. In this case, a declaration with the same name but different argument types makes the virtual functions inaccessible to further derived classes.
W8023: Array variable 'identifier' is near (C++) (see page 367)	(Command-line option to suppress warning: -w-ias) When you use set the Far Data Threshold option, the compiler automatically makes any global variables that are larger than the threshold size be far. When the variable is an initialized array with an unspecified size, its total size is not known when the compiler must decide whether to make it near or far, so the compiler makes it near. The compiler issues this warning if the number of initializers given for the array causes the total variable size to exceed the data size threshold. If the fact that the compiler made the variable... more (see page 367)
W8061: Initialization is only partially bracketed (C++) (see page 367)	(Command-line option to display warning: `-wpin`) When structures are initialized, braces can be used to mark the initialization of each member of the structure. If a member itself is an array or structure, nested pairs of braces can be used. This ensures that the compiler's idea and your idea of what value goes with which member are the same. When some of the optional braces are omitted, the compiler issues this warning.
W8038: constant member 'identifier' is not initialized (C++) (see page 367)	(Command-line option to suppress warning: -w-nci) This C++ class contains a constant member 'member' that doesn't have an initialization. Note that constant members can be initialized only; they can't be assigned to.
W8071: Conversion may lose significant digits (C++) (see page 367)	(Command-line option to display warning: -wsig) For an assignment operator or some other circumstance, your source file requires a conversion from a larger integral data type to a smaller integral data type where the conversion exists. Because the integral data type variables don't have the same size, this kind of conversion might alter the behavior of a program.
W8043: Macro definition ignored (C++) (see page 368)	(Command-line option to suppress warning: -w-nma) An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.
W8017: Redefinition of 'x' is not identical (C++) (see page 368)	(Command-line option to suppress warning: -w-dup) Your source file redefined the macro 'ident' using text that was not exactly the same as the first definition of the macro. The new text replaces the old.
W8079: Mixing pointers to different 'char' types (C++) (see page 368)	(Command-line option to display warning: -wucp) You converted a signed char pointer to an unsigned char pointer, or vice versa, without using an explicit cast. (Strictly speaking, this is incorrect, but it is often harmless.)
W8067: Both return and return with a value used (C++) (see page 368)	(Command-line option to suppress warning: -w-ret) The current function has return statements with and without values. This is legal C, but almost always generates an error. Possibly a return statement was omitted from the end of the function.
W8048: Use qualified name to access member type 'identifier' (C++) (see page 368)	(Command-line option to suppress warning: -w-nst) In previous versions of the C++ specification, `typedef` and tag names declared inside classes were directly visible in the global scope. In the latest specification of C++, these names must be prefixed with class`::`qualifier if they are to be used outside of their class scope. The compiler issues this warning whenever a name is uniquely defined in a single class. The compiler permits this usage without class`::`. This allows older versions of code to compile.

W8039: Constructor initializer list ignored (C++) (see page 368)	(Command-line option to suppress warning: -w-ncl) An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.
W8040: Function body ignored (C++) (see page 369)	(Command-line option to suppress warning: -w-nfd) An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.
W8042: Initializer for object 'x' ignored (C++) (see page 369)	(Command-line option to suppress warning: -w-nin) An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.
W8044: #undef directive ignored (C++) (see page 369)	(Command-line option to suppress warning: -w-nmu) An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.
W8037: Non-const function 'function' called for const object (C++) (see page 369)	(Command-line option to suppress warning = -w-ncf) A non-const member function was called for a const object. (This is an error, but was reduced to a warning to give existing programs a chance to work.)
W8051: Non-volatile function 'function' called for volatile object (C++) (see page 369)	(Command-line option to suppress warning: -w-nvf) In C++, a class member function was called for a volatile object of the class type, but the function was not declared with volatile following the function header. Only a volatile member function can be called for a volatile object. For example, if you have
W8019: Code has no effect (C++) (see page 370)	(Command-line option to suppress warning: -w-eff) This warning is issued when the compiler encounters a statement with some operators that have no effect. For example, the statement
W8057: Parameter 'parameter' is never used (C++) (see page 370)	(Command-line option to suppress warning: -w-par) The named parameter, declared in the function, was never used in the body of the function. This might or might not be an error and is often caused by misspelling the parameter. This warning can also occur if the identifier is redeclared as an automatic (local) variable in the body of the function. The parameter is masked by the automatic variable and remains unused.
W8070: Function should return a value (C++) (see page 370)	(Command-line option to suppress warning: `-w-rvl`) This function was declared (maybe implicitly) to return a value. The compiler found a return statement without a return value, or it reached the end of the function without finding a return statement. Either return a value or change the function declaration to return void.
W8047: Declaration of static function function ignored (C++) (see page 370)	(Command-line option to suppress warning: -w-nsf) An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.
W8041: Negating unsigned value (C++) (see page 370)	(Command-line option to suppress warning: -w-ngu) Basically, it makes no sense to negate an unsigned value because the result will still be unsigned.
W8054: Style of function definition is now obsolete (C++) (see page 371)	(Command-line option to suppress warning = -w-ofp) In C++, this old C style of function definition is illegal:
W8025: Ill-formed pragma (C++) (see page 371)	(Command-line option to suppress warning: -w-ill) A pragma does not match one of the pragmas expected by the compiler.
W8063: Overloaded prefix operator 'operator' used as a postfix operator (C++) (see page 371)	(Command-line option to suppress warning: -w-pre) The C++ specification allows you to overload both the prefix and postfix versions of the ++ and -- operators. Whenever the prefix operator is overloaded, but is used in a postfix context, the compiler uses the prefix operator and issues this warning. This allows older code to compile.
W8015: Declare 'type' prior to use in prototype (C++) (see page 371)	(Command-line option to suppress warning: -w-dpu) When a function prototype refers to a structure type that has not previously been declared, the declaration inside the prototype is not the same as a declaration outside the prototype. For example,
W8069: Nonportable pointer conversion (C++) (see page 372)	(Command-line option to suppress warning: `-w-rpt`) A nonzero integral value is used in a context where a pointer is needed or where an integral value is needed; the sizes of the integral type and pointer are the same. Use an explicit cast if this is what you really meant to do.

W8066: Unreachable code (C++) (see page 372)	(Command-line option to suppress warning: -w-rch) A break, continue, goto, or return statement was not followed by a label or the end of a loop or function. The compiler checks while, do, and for loops with a constant test condition, and attempts to recognize loops that can't fall through.
W8029: Temporary used for parameter '???' (C++) (see page 372)	(Command-line option to suppress warning: -w-lvc) In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter. The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused. In the following example, function f requires a reference to an int, and c is a char:
W8031: Temporary used for parameter 'parameter' OR W8029 Temporary used for parameter 'number' OR W8030 Temporary used for parameter 'parameter' in call to 'function' OR W8032 Temporary used for parameter 'number' in call to 'function' (C++) (see page 372)	(Command-line option to suppress warning: -w-lvc) In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter. The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.
W8032: Temporary used for parameter 2 in call to '???' (C++) (see page 373)	(Command-line option to suppress warning: -w-lvc) In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter. The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused. In the following example, function f requires a reference to an int, and c is a char:
W8028: Temporary used to initialize 'identifier' (C++) (see page 373)	(Command-line option to suppress warning: -w-lin) In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter. The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused. *Example* In this example, function f requires a reference to an int, and c is a char:
W8074: Structure passed by value (C++) (see page 373)	(Command-line option to display warning: -wstv) This warning is generated any time a structure is passed by value as an argument. It is a frequent programming mistake to leave an address-of operator (&) off a structure when passing it as an argument. Because structures can be passed by value, this omission is acceptable. This warning provides a way for the compiler to warn you of this mistake.
W8011: Nonportable pointer comparison (C++) (see page 374)	(Command-line option to suppress warning: -w-cpt) Your source file compared a pointer to a non-pointer other than the constant 0. You should use a cast to suppress this warning if the comparison is proper.
W8075: Suspicious pointer conversion (C++) (see page 374)	(Command-line option to suppress warning: -w-sus) The compiler encountered some conversion of a pointer that caused the pointer to point to a different type. You should use a cast to suppress this warning if the conversion is proper. A common cause of this warning is when the C compiler converts a function pointer of one type to another (the C++ compiler generates an error when asked to do that). It can be suppressed by doing a typecast. Here is a common occurrence of it for Windows programmers:
W8059: Structure packing size has changed (C++) (see page 374)	(Command-line option to suppress warning: -w-pck) This warning message is issued when the structure alignment is different after including a file than it was before including that file. The intention is to warn you about cases where an include file changes structure packing, but by mistake doesn't restore the original setting at the end. If this is intentional, you can give a #pragma nopackwarning directive at the end of an include file to disable the warning for this file. The warning can be disabled altogether by #pragma warn -pck.

W8045: No declaration for function 'function' (C++) (see page 374)	(Command-line option to display warning: -wnod) This message is given if you call a function without first declaring that function. In C, you can declare a function without presenting a prototype, as in
W8073: Undefined structure 'structure' (C++) (see page 375)	(Command-line option to display warning = `-wstu`) Your source file used the named structure on some line before where the error is indicated (probably on a pointer to a structure) but had no definition for the structure. This is probably caused by a misspelled structure name or a missing declaration.
W8013: Possible use of 'identifier' before definition (C++) (see page 375)	(Command-line option to display warning: -wdef) Your source file used the variable 'identifier' in an expression before it was assigned a value. The compiler uses a simple scan of the program to determine this condition. If the use of a variable occurs physically before any assignment, this warning will be generated. Of course, the actual flow of the program can assign the value before the program uses it.
W8004: 'identifier' is assigned a value that is never used (C++) (see page 375)	(Command-line option to suppress warning: -w-aus) The variable appears in an assignment, but is never used anywhere else in the function just ending. The warning is indicated only when the compiler encounters the closing brace. The `#pragma warn -aus` switch has function-level granularity. You cannot turn off this warning for individual variables within a function; it is either off or on for the whole function.
W8081: Void functions may not return a value (C++) (see page 375)	(Command-line option to suppress warning: -w-voi) Your source file declared the current function as returning void, but the compiler encountered a return statement with a value. The value of the return statement will be ignored.
W8078: Throw expression violates exception specification (C++) (see page 376)	(Command-line option to suppress warning: -w-thr) This warning happens when you add an exception specification to a function definition and you throw a type in your function body that is not mentioned in your exception specification. The following program would generate this warning:
W8021: Handler for 'type1' hidden by previous handler for 'type2' (C++) (see page 376)	(Command-line option to suppress warning: -w-hch) This warning is issued when a handler for a type 'D' that is derived from type 'B' is specified after a handler for B', since the handler for 'D' will never be invoked.
W8056: Integer arithmetic overflow (C++) (see page 376)	The compiler detected an overflow condition in an integer math expression. For example:
W8035: User-defined message (C++) (see page 376)	The error message for which you have requested Help is a user-defined warning. In C++ code, user-defined messages are introduced by using the `#pragma` message compiler syntax. `Note:` In addition to messages that you introduce with the `#pragma` message compiler syntax, user-defined warnings can be introduced by third party libraries. Should you require Help about a third party warning, please contact the vendor of the header file that issued the warning.
W8049: Use '> >' for nested templates Instead of '>>' (C++) (see page 376)	(Command-line option to suppress warning: -w-ntd) Whitespace is required to separate the closing ">" in a nested template name, but since it is an extremely common mistake to leave out the space, the compiler accepts a ">>" with this warning.
W8026: Functions with exception specifications are not expanded inline (C++) (see page 377)	*Also:Functions taking class by value arguments are not expanded inline* (Command-line option to suppress warning: -w-inl) Exception specifications are not expanded inline: Check your inline code for lines containing exception specification. Functions taking class-by-value argument(s) are not expanded inline: When exception handling is enabled, functions that take class arguments by value cannot be expanded inline. `Note:` Functions taking class parameters by reference are not subject to this restriction.

W8058: Cannot create pre-compiled header: 'reason' (C++) (see page 377)	(Command-line option to suppress warning: -w-pch) This warning is issued when pre-compiled headers are enabled but the compiler could not generate one, for one of the following reasons: *ReasonExplanation* write failedThe compiler could not write to the pre-compiled header file. This occurs if you specified an invalid location to cache precompiled headers or if the disk is full. code in headerOne of the headers contained a non-inline function body. initialized data in headerOne of the headers contained a global variable definition (in C, a global variable with an initializer; in C++ any variable not declared as 'extern'). header incompleteThe... more (see page 377)
W8046: Pragma option pop with no matching option push (C++) (see page 377)	The compiler encountered a #pragma option pop before a previous #pragma option push, or in the case of nesting, there are more occurrences of #pragma option pop than of #pragma option push.
W8050: No type OBJ file present; disabling external types option. (C++) (see page 377)	(Command-line option to suppress warning: -w-nto) A precompiled header file references a type object file, but the type object file cannot be found. This is not a fatal problem but will make your object files larger than necessary.
W8027: Functions containing 'statement' are not expanded inline (C++) (see page 378)	(Command-line option to suppress warning: -w-inl) *Where:* 'statement' can be any of the following: • Static variables • Aggregate initializers • Some return statements • Local destructors • Some if statements • Local classes • Missing return statements • Disallowed reserved words listed under "Reserved words" below. *Reserved words* Functions containing any of these reserved words can't be expanded inline, even when specified as inline:
W8036: Non-ANSI keyword used: 'keyword' (C++) (see page 378)	(Command-line option to display warning: -wnak) A non-ANSI keyword (such as '__fastcall') was used when strict ANSI conformance was requested via the -A option.
W8053: 'ident' is obsolete (C++) (see page 379)	(Command-line option to suppress warning: -w-obs) Issues a warning upon usage for any "C" linkage function that has been specified. This will warn about functions that are "obsolete". Here's an example of it's usage:
W8103: Path 'path' and filename 'filename' exceed maximum size of 'n' (C++) (see page 379)	(Command-line option to display warning: -wstv) In looking up include files, the C++ compiler has encountered a file whose path and filename contain more characters than are allowed in the Windows maximum. Rename the paths and filenames that you can, and shorten their names wherever possible.
W8062: Previous options and warnings not restored (C++) (see page 379)	The compiler didn't encounter a #pragma option pop after a previous #pragma option push, or in the case of nesting, there are more occurrences of #pragma option push than of #pragma option pop.
W8055: Possible overflow in shift operation (C++) (see page 379)	The compiler detects cases where the number of bits shifted over is larger than the number of bits in the affected variable; for example:
W8072: Suspicious pointer arithmetic (C++) (see page 379)	This message indicates an unintended side effect to the pointer arithmetic (or array indexing) found in an expression.

W8033: Conversion to 'type' will fail for members of virtual base 'class' (C++) (see page 380)	(Command-line option to suppress warning: -w-mpc) This warning is issued only if the `-Vv` option is in use. The warning may be issued when a member pointer to one type is cast to a member pointer of another type and the class of the converted member pointer has virtual bases. Encountering this warning means that at runtime, if the member pointer conversion cannot be completed, the result of the cast will be a NULL member pointer.
W8034: Maximum precision used for member pointer type 'type' (C++) (see page 380)	(Command-line option to suppress warning: -w-mpd) When a member pointer type is declared, its class has not been fully defined, and the `-Vmd` option has been used, the compiler has to use the most general (and the least efficient) representation for that member pointer type. This can cause less efficient code to be generated (and make the member pointer type unnecessarily large), and can also cause problems with separate compilation; see the `-Vm` compiler switch for details.
E2537: Cannot create instance of abstract class (C++) (see page 380)	This class is an abstract class, which you cannot instantiate.
E2018: Cannot catch 'identifier' -- ambiguous base class 'identifier' (C++) (see page 380)	It is not legal to catch a class that contains more than one copy of a (non-virtual) base class. However, you can catch the exception as a more derived type. For example:
E2550: No arguments can follow a variadic template in an argument list (C++) (see page 381)	In an argument list, the variadic template must not be followed by arguments.
E2538: Static assert failed: '%s' (C++) (see page 381)	See Static Assertions (see page 498) for details about how to use static_assert., which is one of the C++0x features.
E2548: ... was unexpected; expression is not a variadic template pattern (C++) (see page 381)	This message pertains to the expected syntax of variadic templates.
E2543: Combination of options 'options' is not permitted (C++) (see page 381)	Eliminate the combination of options. For example, you cannot combine the C++ compiler options -b- and -bi because -b- turns off all -b options, including -bi.
E2549: Operand is not a parameter pack (C++) (see page 381)	A parameter pack is required in the indicated location. This message pertains to variadic templates.
E2544: Function exception specifications do not match (C++) (see page 381)	The throw specifications on two function declaration/definitions which refer to the same function do not agree. Rewrite the function exception specifications to match.
E2536: Incomplete type cannot be part of a exception declaration (C++) (see page 381)	This error occurs when an incomplete type is used in exception declaration. From C++ ISO 14882:1998 15.3.1: "The exception-declaration shall not denote an incomplete type. The exception-declaration shall not denote a pointer or reference to an incomplete type, other than void*, const void*, volatile void*, or const volatile void*. Types shall not be defined in an exception-declaration.
E2535: Incomplete type cannot be part of a throw specification (C++) (see page 382)	This error occurs when an incomplete type is used in a throw specification. From C++ ISO 14882:1998 15.1.3: "The type of the throw-expression shall not be an incomplete type, or a pointer or reference to an incomplete type, other than void*, const void*, volatile void*, or const volatile void*." This includes exception specifications (15.4.1).
E2532: Constant in new expression requires an initializer (C++) (see page 382)	If the type is const then either an initializer must be present or the non-POD class must have a user-declared constructor. Error in ANSI mode, warning if ANSI mode is not set.
E2541: Attribute '%s' cannot be set (C++) (see page 382)	This error occurs if the specified attribute cannot be applied to this entity.
E2545: Enum underlying type must be an integral (C++) (see page 382)	Ensure that the underlying type of the enumeration is an integral type. The underlying type of a scoped enum is fixed to int unless explicitly provided by the user. The underlying type of a 'classic' enum remains unspecified unless fixed by the user. For more information, see Strongly Typed Enums (see page 498).
E2546: Redeclaration of enum is not identical (C++) (see page 382)	This error occurs if: • a re-declaration of an enum has a different underlying type • a scoped enum is re-declared as an unscoped enum • an unscoped enum is re-declared as a scoped enum For more information, see Strongly Typed Enums (see page 498).
E2533: Parameter mismatch (wanted typename) (C++) (see page 382)	This error occurs when a parameter passed to a type trait function is not the correct type.

E2534: Integral constant expression expected (C++) (see page 382)	Integral constant expressions are described in section 5.19 of the C++ standard, and are sometimes referred to as "compile time constants". A working draft for the C++ standard can be found at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf.
E2531: Parameter is an incomplete type (C++) (see page 383)	This error occurs when an incomplete type is passed into a type trait function that does not accept incomplete type parameters. For example:
E2539: Constant expression expected (C++) (see page 383)	You need to supply a constant expression in the indicated location. For details about using a constant expression in a static assertion, see Static Assertions (see page 498).
E2547: ... expected (C++) (see page 383)	An ellipsis (...) is expected in the location indicated.
E2540: String literal expected (C++) (see page 383)	You need to specify a string literal in the position indicated in the error message. A string literal is the second argument in a static assertion. For details about using a static assertion, see Static Assertions (see page 498).
E2552: This feature is not (yet) supported (C++) (see page 383)	Support might be added in a coming release.
E2542: '%s' is marked 'final' and cannot be overriden (C++) (see page 383)	The attribute 'final' applies to class definitions and to virtual member functions being declared in a class definition. If the attribute is specified for a class definition, it is equivalent to being specified for each virtual member function of that class, including inherited member functions. If a virtual member function f in some class B is marked 'final' and in a class D derived from B, a function D::f overrides B::f, the program is ill-formed.
E2553: %s mismatch in redeclaration of '%s' (C++) (see page 383)	Attributes specified on a declaration have to be a subset of attributes specified on a definition.
E2551: Return statement not allowed in __finally block (C++) (see page 384)	Rewrite the `__finally` block so that it does not contain a return statement. For more information, see Writing a finally Block in C++ (see page 2023) and finally keyword (see page 550).
W8104: Local Static with constructor dangerous for multi-threaded apps (C++) (see page 384)	(Command-line option to suppress warning: -w-mls) This warning is generated for local static objects with constructors for multithreaded programs. This situation can cause problems if two threads enter the containing function at the same time and no critical sections are present, allowing the constructor to potentially be called more than once.
W8106: %s are deprecated (C++) (see page 384)	You have used an old style syntax or language use that is deprecated -- that is, no longer recommended and might be phased out in the future.
W8110: Duplicate '%s' attribute directive ignored (C++) (see page 384)	(Command-line option to suppress warning: -w-dat) This warning is generated if the same attribute was specified more than once for the same entity.
W8108: Constant in new expression requires an initializer (C++) (see page 384)	(Command-line option to suppress warning: -w-nci) If the type is const then either an initializer must be present or the non-POD class must have a user-declared constructor. Error in ANSI mode, warning if ANSI mode is not set.
W8113: Inline function was declared with 'extern template' (C++) (see page 384)	(Command-line option to suppress warning: -w-iex) 'extern template' has no normative effect on inline functions. Implementations are encouraged to suppress out-of-line copies of inline functions that were declared with 'extern template'.
W8109: Parameter '%s' is a dependent type (C++) (see page 385)	(Command-line option to enable warning: -wpad) This warning is generated if a parameter of an intrinsic function is a dependent type and the dependency cannot be resolved.
W8105: Reference/Constant member 'identifier' in class without constructors (C++) (see page 385)	(Command-line option to suppress warning: -w-mnc) This warning is generated for a reference or constant member in a class without constructors if ANSI mode is not set. In ANSI mode this is an error.
W8107: Type name expected (C++) (see page 385)	(Command-line option to enable warning: -wntn) This warning is generated if the type of a class/union member is omitted in its declaration and ANSI mode is not set. Always an error in ANSI mode.
W8112: Unresolved dependencies in expression (C++) (see page 385)	(Command-line option to suppress warning: -w-dex) This warning is generated if dependencies in the expression cannot be resolved.

3.1.2.1 E2066: Invalid MOM inheritance (C++)

The compiler issues this error if the currently compiled class doesn't have the same MOM (Microsoft Object Model) related flags set as its direct parent.

This compiler error message is deprecated.

3.1.2.2 E2525: You must define _PCH_STATIC_CONST before including xstring to use this feature (C++)

You attempted to use a feature defined in xstring, part of the Dinkumware standard C++ library. The C++ compiler could not generate a precompiled header because there is a constant (defined in xstring) in the header. If you want to include xstring, you should first set the define _PCH_STATIC_CONST.

3.1.2.3 E2526: Property 'name' uses another property as getter/setter; Not allowed (C++)

Properties typically have both a getter and a setter, but a property cannot serve as either the getter or setter of another property.

3.1.2.4 E2008: Published property access functions must use __fastcall calling convention (C++)

The calling convention for access functions of a property (read, write, and stored) declared in a __published section must be __fastcall. This also applies to hoisted properties.

Example

```
struct__declspec(delphiclass) clxclass{int__fastcall Getter1(void); int__cdecl
Getter2(void); __published:int __property ip1 = {read = Getter1};    // OKint __property ip2 =
{read = Getter2};    // Error};
```

3.1.2.5 E2122: Function call terminated by unhandled exception 'value' at address 'addr' (C++)

This message is emitted when an expression you are evaluating while debugging includes a function call that terminates with an unhandled exception. For example, if in the debugger's evaluate dialog, you request an evaluation of the expression foo()+1 and the execution of the function foo() causes a GP fault, this evaluation produces the above error message.

You may also see this message in the watches window because it also displays the results of evaluating an expression.

3.1.2.6 E2506: Explicit specialization of 'specifier' is ambiguous: must specify template arguments (C++)

In the following code, explicit template arguments are necessary:

```
template<class T> void foo(T);
template<class T> void foo(T *);
template<> void foo(int *); // error, must say 'foo<int>' or 'foo<int *>'
```

3.1.2.7 E2483: Array dimension 'specifier' could not be determined (C++)

If, during instantiation of a type, an array dimension cannot be computed—usually this is due to some other error which would be reported—then this error will result.

For example, if an array dimension is dependent upon a template parameter but an error occurs while it is being parsed and the template argument being substituted does not yield a legal constant expression, this error is displayed.

3.1.2.8 E2509: Value out of range (C++)

The inline assembler has detected a numeric overflow in one of your expressions. Make sure all of your numbers can fit in 32 bits.

3.1.2.9 E2510: Operand size mismatch (C++)

Help is not available for this item.

3.1.2.10 E2050: __declspec(delphireturn) class 'class' must have exactly one data member (C++)

This is an internal compiler error. A class marked as a delphireturn class has more than one non-static data member.

3.1.2.11 E2530: Unrecognized option, or no help available (C++)

You have entered a command element that the C++ compiler cannot parse, or the option you entered has no associated help. Try again.

3.1.2.12 E2527: Option 'name' cannot be set via 'name' (C++)

An attempt was made in a source file to set an option that cannot be set after either parsing or compiling of the file starts. Instead, set this option on the command line or in a `.cfg` file.

For example, if a source file contains a `#pragma option push –v`, you need to remove the push or set /unset this option either on the command line or in a `.cfg` file.

3.1.2.13 E2528: Option 'name' must be set before compilation begins (C++)

An attempt was made in a source file to set an option that must be set before compiling starts. Instead, set this option on the command line, in a `.cfg` file, or at the top of the source file before the line `int foo();` .

3.1.2.14 E2074: Value after -g or -j should be between 0 and 255 inclusive (C++)

Both the -g and the -j command line options can be followed by an optional number. The compiler expects this number to be between 0 and 255 inclusive.

3.1.2.15 E2492: Properties may only be assigned using a simple statement, e.g. \"prop = value;\" (C++)

Assignments to properties should be made in simple assignment statements. If property assignments could become Lvalues, which happens when property assignments are embedded in larger statements, the getter is called to create the Lvalue, with all the side effects that getter causes. The compiler allows only one call to either the getter or the setter in a statement.

For example:

```
class myClass
{
  int X;
  public:
  int __property x = { read=getx, write=putx };
  int getx() { return X; }
  void putx(int val) { X = val; }
} OneClass;
int value(int);
int main()
{
  return value(OneClass.x = 4);  // This causes an error
}
```

3.1.2.16 E2505: Explicit instantiation requires an elaborated type specifier (i.e.,"class foo<int>") (C++)

The following code is illegal:

```
template<class T> class foo;
template foo<int>;            // missing `class' keyword
```

See Also

Implicit And Explicit Template Functions (see page 434)

3.1.2.17 E2100: Invalid template declarator list (C++)

It is illegal for a declarator list to follow a template class declaration. For example:

```
template<class T>
class foo {
} object_name;  // This causes an error
```

3.1.2.18 E2102: Cannot use template 'template' without specifying specialization parameters (C++)

The generic form of a template must be referenced using specialization parameters. For example, for a template class named foo, taking two template parameters, then a legal reference might have the form

```
foo<int, char>
```

Referring to the template as just foo is legal in only two circumstances:

- When passing the template name as a template template argument
- While declaring the members of that class template, to refer to the enclosing template type

For example:

```
template<class T>
class foo
{
public:
    foo();       // legal use of bare template name
    foo& operator=(const foo&);
};
foo<foo> x;   // error: not a template template argument
foo y;    // error: needs specialization parameters
```

3.1.2.19 E2107: Invalid use of template 'template' (C++)

This error results when attempting to use a template template parameter in any way other than to reference a template specialization, or to pass that parameter in turn as a template template argument to another template. For example:

```
template<template<class T> class U>
class foo;
template<template<class T> class U>
class bar
{
U x;// error: not a specialization
U<U> y;// ok: used as a specialization, and as a
// template template argument
U<bar> z;// ok: used to reference a specialization
};
```

3.1.2.20 E2105: 'template' qualifier must specify a member template name (C++)

When parsing code that depends in some way upon a template parameter, it is sometimes impossible to know whether a member name will resolve to a template function name, or a regular parameter. In the following code, a 'template' qualifier is required in order to know if the '<' (less-then) operator should be parsed as the beginning character of a template argument list, or as a regular less-than operator:

```
template<class T>
void foo(T a)
{
a.member<10>();
}
```

Although it may be apparent to the reader what is meant, the compiler does not know that "member" refers to a member template function, and it will parse the line of code as follows:

```
a.member < (10>());
```

In order to tell the compiler that the less-than character begins a template argument list, the 'template' qualifier is needed:

```
a.template member<10>();   // "member" must be a member template
```

If the 'template' qualifier is used in a situation where "member" does not resolve to a member template, the above error will result.

3.1.2.21 E2066: Information not available (C++)

Help is not available for this item.

3.1.2.22 E2471: pragma checkoption failed: options are not as expected (C++)

You can use #pragma checkoption to check that certain switches are in the state that you expect. If #pragma checkoption detects that a switch is not in the expected state, the compiler displays this error.

You can use the following syntax:

```
#pragma checkoption <options>
```

For example:

```
#pragma checkoption -O2
```

OR

```
#pragma checkoption -C -O2
```

The compiler will check if the option(s) are turned on. If all are turned on, nothing happens. If at least one is not turned on, this error is displayed.

3.1.2.23 E2504: 'dynamic' can only be used with non-template member functions (C++)

You tried to use dynamic with a template member function. Dynamic functions are allowed for classes derived from TObject. Dynamic functions occupy a slot in every object that defines them, not in any descendants. That is, dynamic functions are virtual functions stored in sparse virtual tables. If you call a dynamic function, and that function is not defined in your object, the virtual tables of its ancestors are searched until the function is found.

3.1.2.24 E2191: '__far16' may only be used with '__pascal' or '__cdecl' (C++)

This is an internal compiler error. The compiler emits this message if the keyword __far16 is mixed with one of the keywords __pascal or __cdecl, all in the same declaration.

3.1.2.25 E2199: Template friend function 'function' must be previously declared (C++)

Not used

3.1.2.26 E2502: Error resolving #import: problem (C++)

Where `problem` can be any of the following relating to problems with the various attributes of #include (see page 692):

unexpected import directive value attribute 'attribute'A value was supplied for the indicated attribute. None was expected.

missing ')' in import directive attribute 'attribute'The value for the indicated attribute was incorrectly specified : a closing parenthesis is missing.

unrecognized import directive attribute 'attribute'The indicated token is not a legitimate attribute for the #import directive.

invalid values for raw_property_prefixes attributeThe values for the raw_property_prefixes attribute were incorrectly specified.

unexpected duplicate property 'property'The indicated #import attribute was specified more than once -- this is an error.

unexpected duplicate get method for property 'property'The get method for the indicated property was specified more than once.

unexpected duplicate put method for property 'property'The put method for the indicated property was specified more than once.

unexpected duplicate put-reference method for property 'property'The put-reference method for the indicated property was specified more than once.

no return value specified for property get method 'method'The indicated property get method does not supply the correct return type.

no return value specified for property put method 'method'The indicated property put method does not supply the correct return type.

could not load type library in 'filename'The indicated type library could not be opened.

could not obtain type library nameThe compiler could not obtain obtain a library name for the current type library

See Also

#include (see page 692)

3.1.2.27 E2501: Unable to open import file 'filename' (C++)

This error occurs when you use:

```
#import "somefile.h"
```

and the file you are trying to import doesn't exist or can't be found by the compiler.

3.1.2.28 E2494: Unrecognized __declspec modifier (C++)

A _declspec modifier was given that is not valid.

3.1.2.29 E2493: Invalid GUID string (C++)

The GUID string does not have the form of a Globally Unique Identifier.

3.1.2.30 E2499: Invalid __declspec(uuid(GuidString)) format (C++)

This error happens when you used the wrong format to define your GuidString. GUIDs are defined for structs in the following way:

```
class __declspec(uuid("19a76fe0-7494-11d0-8816-00a0c903b83c")) foo{};
```

You would get the above mentioned error for instance from:

```
class __declspec(uuid(19a76fe0-7494-11d0-8816-00a0c903b83c)) foo{}; //Missing quotes
```

or

```
class __declspec(uuid"7494-11d0-8816-00a0c903b83c")) foo{}; // Missing left parentheses
```

3.1.2.31 E2496: Invalid call to uuidof(struct type|variable) (C++)

The uuidof operator was given an incorrect argument.

3.1.2.32 E2511: Unterminated macro argument (C++)

A macro argument that was started on the line listed has not been properly terminated

3.1.2.33 E2489: Maximum option context replay depth exceeded; check for recursion (C++)

If this error is triggered, it means that recursive template instantiation has gone too deep. Check for compile-time recursion in your program, and limit it to no more than 256 levels.

3.1.2.34 E2488: Maximum token reply depth exceeded; check for recursion (C++)

If this error is triggered, it means that recursive template instantiation has gone too deep. Check for compile-time recursion in your program, and limit it to no more than 256 levels.

3.1.2.35 E2491: Maximum VIRDEF count exceeded; check for recursion (C++)

Too many VIRDEF symbols were allocated. The compiler imposes a limit to the number of VIRDEFs allowed per translation unit. Currently this limit is in the order of 16384 VIRDEFs.

One way this could happen is if a program has more than 16384 functions.

3.1.2.36 E2230: In-line data member initialization requires an integral constant expression (C++)

Static const class members, which are initialized in the body of the class, have to be initialized with a constant expression of integral type.

3.1.2.37 E2241: VCL style classes need virtual destructors (C++)

Destructors defined in VCL style classes have to be virtual.

Example

```
struct __declspec(delphiclass) vclclass1
{
~vclclass1() {}           // Error
};
struct __declspec(delphiclass) vclclass2
{
virtual ~vclclass2() {}   // OK
};
```

3.1.2.38 E2524: Anonymous structs/unions not allowed to have anonymous members in C++ (C++)

The C++ compiler requires that the members of an anonymous struct or union be named.

3.1.2.39 E2246: x is not abstract public single inheritance class hierarchy with no data (C++)

Internal compiler error. In some cases, the compiler will enforce restrictions on a class hierarchy. In this case, the restrictions would be that all classes are abstract classes, and all classes only have one parent.

3.1.2.40 E2249: = expected (C++)

The compiler expected an equal sign in the position where the error was reported but there was none. This is usually a syntax error or typo.

3.1.2.41 E2267: First base must be VCL class (C++)

Internal compiler error. In some cases, the compiler will enforce restrictions on a class hierarchy. In this case, the restrictions would be that the first parent of a class is a Delphi style class.

3.1.2.42 E2472: Cannot declare a member function via instantiation (C++)

If a declaration within a template class acquires a function type through a type dependent on a template-parameter and this results in a declaration that does not use the syntactic form of a function declarator to have function type, the program is ill-formed. For example:

```
template<class T>
struct A {
static T t;
};
typedef int function();
A<function> a;// error: would declare A<function>::t
// as a static member function
```

Another example:

In the example below, the template member 'a' has type 'T'. If the template is instantiated with T as a function type, it implies that 'a' is therefore a member function. This is not allowed and the error message is displayed.

```
template<T& x> class foo { T a; }
int func(int);
template class foo<func>;
```

3.1.2.43 E2515: Cannot explicitly specialize a member of a generic template class (C++)

You are trying to make a generic template into a specialized member. For example, the following code is illegal:

```
    template<typename T>
class foo {
template<typename U>
      class bar {
      };
    };
template<typename T>
template<>
class foo<T>::bar<char> {
};
```

The second declaration in the example is an error, because it tries to explicitly specialize bar<char> within foo<T>.

3.1.2.44 E2474: 'function' cannot be declared as static or inline (C++)

You attempted to declare a symbol as static or inline and this type of symbol cannot be defined as static or inline. Certain functions, like 'main' and 'WinMain' cannot be declared static or inline. 'main' is the entrypoint of console applications, and 'WinMain' is the entry point of Windows applications.

For example, this error is displayed in the following cases:

```
static int main()  // This causes an error
{}
```

or

```
inline int main() { return 0; }
```

3.1.2.45 E2498: Need previously defined struct GUID (C++)

This happens when you use the __uuidof operator without including a header that defines the GUID struct. So the following program code would display this error:

```
class __declspec(uuid("19a76fe0-7494-11d0-8816-00a0c903b83c")) foo{};
int main()
{
    __uuidof(foo);
    return 0;
}
```

And you would fix it as follows:

```
#include <windows.h>    // Will pull in struct GUID
class __declspec(uuid("19a76fe0-7494-11d0-8816-00a0c903b83c")) foo{};
int main()
{
    __uuidof(foo);
    return 0;
}
```

3.1.2.46 E2295: Too many candidate template specializations from 'specifier' (C++)

When reference a class template specialization, it is possible that more than one possible candidate might result from a single reference. This can only really happen among class partial specializations, when more than one partial specialization is contending for a possible match:

```
template<class T, class U>
class foo;
template<class T>
class foo<T, T *>;
template<class T>
class foo<T *, T>;
foo<int *, int *> x;  // error: which partial specialization to use?
```

In this example, both partial specializations are equally valid, and neither is more specialized than the other, so an error would result.

3.1.2.47 E2475: 'function' cannot be a template function (C++)

Certain functions, like 'main' and 'WinMain' cannot be declared as a template function. 'main' is the entrypoint of console applications, and 'WinMain' is the entry point of Windows applications.

For example:

```
template <class T> int main()  // This causes an error
{}
```

See Also

Function Templates (see page 433)

3.1.2.48 E2299: Cannot generate template specialization from 'specifier' (C++)

This error will result if an attempt is made to reference a template class or function in a manner which yields no possible candidate specializations. For example:

```
template<class T>
class foo;
foo<10> x;       // error: arguments aren't valid for 'foo'
```

3.1.2.49 E2300: Could not generate a specialization matching type for 'specifier' (C++)

This error is no longer generated by the compiler.

3.1.2.50 E2497: No GUID associated with type:'type' (C++)

A variable or type was used in a context requiring a GUID, but the type does not have a GUID associated with it. GUIDs are associated with types using _declspec (uuid(GUID)).

3.1.2.51 E2522: Non-const function 'function' called for const object (C++)

Data type mismatch resulting in an erroneous function call. The object of the call (a non-const function) should be a const object.

3.1.2.52 E2523: Non-volatile function 'name' called for volatile object (C++)

Data type mismatch. The error is the result of an erroneous function call. The object of the call (a non-volatile function) should be a volatile object.

3.1.2.53 E2513: Cannot emit RTTI for 'parameter' in 'function' (C++)

The compiler issues this error if it cannot generate RTTI information for the return type of a parameter. See Runtime type information for more information.

3.1.2.54 E2512: Cannot emit RTTI for return type of 'function' (C++)

The compiler issues this error if the it cannot generate RTTI information for the return type of a function. See Runtime type information for more information.

3.1.2.55 E2507: 'class' is not a direct base class of 'class' (C++)

The first type is not a direct base class of the second type. A direct base class refers to the immediate derivations of that class, and not the derivations of its subclasses.

3.1.2.56 E2529: Path 'path' exceeds maximum size of 'n' (C++)

In looking up include files, the C++ compiler has encountered a file whose path name contains more characters than are allowed in the Windows maximum. Rename the path to a shorter name.

3.1.2.57 E2495: Redefinition of uuid is not identical (C++)

GUID's attached to structs have to be the same across multiple declarations and definitions of the same struct. So the following example would cause this error:

```
class __declspec(uuid("19a76fe0-7494-11d0-8816-00a0c903b83c")) foo;
class __declspec(uuid("00000000-7494-11d0-8816-00a0c903b83c")) foo{}
```

3.1.2.58 E2500: __declspec(selectany) is only for initialized and externally visible variables (C++)

You cannot use __declspec(selectany) with static variables, unitialized variables, etc.

3.1.2.59 E2482: String constant expected (C++)

The compiler expected a string constant at this location but did not receive one.

This error is no longer generated by the compiler.

3.1.2.60 E2481: Unexpected string constant (C++)

There are times when the compiler does not expect a string constant to appear in the source input. For example:

```
class foo { "Hello"; };
```

3.1.2.61 E2386: Cannot involve parameter 'parameter' in a complex partial specialization expression (C++)

When declaring or defining a template class partial specialization, it is illegal to involve any of the non-type template parameters in complex expressions. They may only be referenced by name. For example:

```
template<class T, int U>
class foo;
template<int U>
class foo<char, U * 3>;// error: "U * 3" is a complex expression
template<int U>
class foo<char, U>;// OK: "U" is a simple, by-name expression
```

3.1.2.62 E2387: Partial specializations may not specialize dependent non-type parameters ('parameter') (C++)

A partial specialization may not use a template parameter in its specialization argument list which is dependent on another type parameter. For example:

```
template<class T, int U>
class foo;
template<class T, T U>
class foo<T *, U>;      // error: 'U' is type-dependent
```

3.1.2.63 E2388: Argument list of specialization cannot be identical to the parameter list of primary template (C++)

When declaring a partial specialization, its specialization argument list must differ in some way from its basic parameter list. For example:

template<class T>

```
class foo;
template<class T>
class foo<T *>;     // OK: is more specialized than primary template
template<class T>
class foo<T>;       // error: identical to primary template
```

3.1.2.64 E2389: Mismatch in kind of substitution argument and template parameter 'parameter' (C++)

When referencing a template specialization, all type parameters must be satisfied using type arguments, all non-type parameters require non-type arguments, and all template template parameters require either a template name, or another template template argument. Mismatching these requirements in any way will trigger the above error. For example:

```
template<class T, int U, template<class V> class W>
class foo;
foo<char, 10, foo> x;    // OK: all parameter kinds match
foo<10, char, int> y;    // error: no parameter kinds match at all!
```

3.1.2.65 E2480: Cannot involve template parameters in complex partial specialization arguments (C++)

A partial specialization cannot reference other template parameters in a nonvalue argument expression, unless it is simply a direct reference to the template parameter. For example:

```
template<int A, int B, int C> class foo;
template<int A> class foo<A+5, A, A+10>;
```

The partial specialization has two illegal arguments. 'A+5' is a complex expression because it uses 'A' in a manner other than as merely a direct argument. The reference to plain 'A' in the second argument is fine, but the third argument is also illegal because it references 'A' in a complex manner as well.

3.1.2.66 E2392: Template instance 'template' is already instantiated (C++)

There are two ways to trigger this error. If –A is enabled (ANSI compliant mode), then attempting to explicitly instantiate a template specialization which has already been instantiated (either implicitly or explicitly) will cause this error. Regardless of –A, attempting to explicitly specialize a template specialization which has already been either implicit or explicitly instantiated will always trigger this error. For example:

```
template<class T>
class foo;
foo<char> x;     // causes implicit instantiation of "foo<char>"
template<>
class foo<char> { };   // error: "foo<char>" already instantiated
template class foo<char>;   // error in –A mode, otherwise a warning
```

3.1.2.67 E2393: Cannot take the address of non-type, non-reference template parameter 'parameter' (C++)

A template parameter has no address, and is not associated with any real "object". Therefore, to take its address, or attempt to assign to it, has no meaning. For example:

```
template<int U>
void foo()
{
int *x = &U;// error: cannot take address of parameter
}
```

3.1.2.68 E2399: Cannot reference template argument 'arg' in template class 'class' this way (C++)

The compiler no longer generates this error.

3.1.2.69 E2397: Template argument cannot have static or local linkage (C++)

Only integral constant expressions, and the address of global variables with external linkage, may be used as template arguments. For example:

```
template<char *x>
class foo;
const char *p = "Hello";
extern char *q;
foo<p> x;// OK: "p" is visible to the outside
foo<q> y;// OK: "q" is also globally visible
foo<"Hello"> z;// error: string literal has static linkage
```

3.1.2.70 E2485: Cannot use address of array element as non-type template argument (C++)

Non-type template arguments may only be of integral type, or the address of a global variable. They cannot be the address of an array element. For example:

```
int x[100];
template<int T>
class foo;
foo<&x[0]> y;// error: not an integral or global address
```

3.1.2.71 E2402: Illegal base class type: formal type 'type' resolves to 'type' (C++)

When instantiating a template class definition, if it is found that a declared base class does not resolve to an accessible class type, this error will result. For example:

```
template<class T>
class foo : public T { };
foo<int> x;// error: "int" is not a valid base class
foo<bar> y;// error: "bar" is an unknown type
```

3.1.2.72 E2403: Dependent call specifier yields non-function 'name' (C++)

The compiler no longer generates this error.

3.1.2.73 E2404: Dependent type qualifier 'qualifier' has no member type named 'name' (C++)

If a template declaration references a member of a dependent type, it is only possible to alert the user to the non-existence of this member during type instantiation for a given set of template arguments. For example:

```
template<class T>
class foo
{
typename T::A x;     // we expect that "A" is a member type
};
struct bar { };
foo<bar> y;// error: "bar" has no member type named "A"
```

3.1.2.74 E2405: Dependent template reference 'identifier' yields non-template symbol (C++)

If a template specialization reference within a template declaration yields a reference to a non-template during type instantiation, the above error will result. For example:

```
template<class T>
class foo
{
```

```
typename T::template A<int> x;  // "A" must be a member template
};
struct bar {
struct A { };
};
foo<bar> y;// error: bar::A is a non-template class!
```

3.1.2.75 E2406: Dependent type qualifier 'qualifier' is not a class or struct type (C++)

If a dependent name reference within a template declaration results in a non-struct member qualification at instantiation time, the above error will result. For example:

```
template<class T>
class foo
{
typename T::A x;     // we expect that "A" is a member type
};
foo<int> y;// error: "int" cannot be qualified; not a class
```

3.1.2.76 E2407: Dependent type qualifier 'qualifier' has no member symbol named 'name' (C++)

If a template declaration references a member of a dependent type, it is only possible to alert the user to the non-existence of this member during type instantiation for a given set of template arguments. For example:

```
template<class T>
class foo
{
foo(int *a = T::A);     // we expect that "A" is a data member
};
struct bar { };
foo<bar> y;// error: "bar" has no member named "A"
```

3.1.2.77 E2408: Default values may be specified only in primary class template declarations (C++)

Template functions, and class partial specializations, may not use default expressions in their template parameter lists. Only primary template declarations may do this. For example:

```
template<class T = int>
class foo;// OK: primary class template
template<class T = int>
void bar();// error: template function
template<class T = int>
class foo<T *>;// error: partial specialization
```

3.1.2.78 E2409: Cannot find a valid specialization for 'specifier' (C++)

This error is no longer generated by the compiler.

3.1.2.79 E2410: Missing template parameters for friend template 'template' (C++)

If a friend template is declared, but no template parameters are specified, this error will result. For example:

```
template<class T>
class foo;
class bar {
friend class foo;// error: forgot template parameters!
};
```

3.1.2.80 E2486: Cannot use address of class member as non-type template argument (C++)

Non-type template arguments may only be of integral type, or the address of a global variable. They cannot be the address of a class member. For example:

```
struct bar {
int x;
} y;
template<int T>
class foo;
foo<&y.x> z;// error: not an integral or global address
```

3.1.2.81 E2411: Declaration of member function default parameters after a specialization has already been expanded (C++)

If a member function of a class template is declared, and then a specialization of that class implicitly instantiated, and later that member function defined with default parameters specified, the above error will result. For example:

```
template<int i>
class foo {
void method(int a, int b = i);
};
foo<10> x;
template<int i>
void foo<i>::method(int a = i, int b);  // error!
```

3.1.2.82 E2412: Attempting to bind a member reference to a dependent type (C++)

The compiler no longer generates this error.

3.1.2.83 E2414: Destructors cannot be declared as template functions (C++)

Destructors cannot be templates. For example:

```
class foo {
template<class T>
virtual ~foo();// error: don't try this at home!
```

```
};
```

3.1.2.84 E2473: Invalid explicit specialization of 'specifier' (C++)

Attempting to explicitly specialize a static data member or any non-template will cause this error.

3.1.2.85 E2490: Specialization within template classes not yet implemented (C++)

Explicit and partial specialization of member template classes and functions within template classes and nested template classes, is not supported.

3.1.2.86 E2416: Invalid template function declaration (C++)

The compiler no longer generates this error.

3.1.2.87 E2417: Cannot specify template parameters in explicit specialization of 'specifier' (C++)

The compiler no longer generates this error.

3.1.2.88 E2418: Maximum instantiation depth exceeded; check for recursion (C++)

The compiler only supports 256 levels of instantiation before it will trigger this error. The main problem is in controlling stack depth, because the parser uses recursive functions to manage type instantiation. Here is an example that would produce such an error:

```
template<int T>
class foo {
public:
static const int x = foo<T – 1>::x;
};
template<int T>
class foo<1> {
public:
static const int x = 1;
};
int main() {
int y = foo<100000>::x;// error: instantiation depth exceeded
}
```

3.1.2.89 E2420: Explicit instantiation can only be used at global scope (C++)

Explicit instantiation cannot be specified at any level other than namespace or global scope. For example:

```
template<class T>
```

```
class foo { };
template class foo<char>;// OK: at global scope
int main() {
template class foo<int>:// error: local scope
}
```

3.1.2.90 E2422: Argument kind mismatch in redeclaration of template parameter 'parameter' (C++)

If a template is declared at one point in the translation unit, and then redeclared with template parameters of a different kind at another location, this error will result. For example:

```
template<class T>
class foo;
// ? time passes ?
template<int T>
class foo;// error: type vs. non-type parameter
```

3.1.2.91 E2423: Explicit specialization or instantiation of non-existing template 'template' (C++)

Attempting to explicit specialize or instantiate a template which does not exist is clearly illegal. For example:

```
template<class T>
class foo;
template class bar<char>;// error: what is "bar"??
template<>
class bar<int> { };// error: there's that "bar" again?
```

3.1.2.92 E2479: Cannot have both a template class and function named 'name' (C++)

No other function or type may have the same name as a template class. For example:

```
void foo();// error: there is a template class named "foo"
template<class T>
class foo;
```

3.1.2.93 E2484: The name of template class 'class' cannot be overloaded (C++)

Attempting to declare a function that overrides the name of a template class will cause this error. For example:

```
template<class T>
class foo;
void foo();// error: there is a template class named "foo"
```

3.1.2.94 E2426: Explicit specialization of 'specifier' requires 'template<>' declaration (C++)

According to the standard, explicit specialization of any template now always require the "template<>" declarator syntax. For example:

```
template<class T>
class foo;
template<>
class foo<char>;// OK: "template<>" was provided
class foo<int>;// error: "template<>" required
```

3.1.2.95 E2487: Cannot specify default function arguments for explicit specializations (C++)

An explicit specialization of a function may not declare default function arguments. For example:

```
template<class T>
void foo(T a);
template<>
void foo<int>(int a = 10);// error: default value not allowed
```

3.1.2.96 E2427: 'main' cannot be a template function (C++)

'main' cannot be declared as a template function. 'main' is the entry point of a console application, and it should be declared as a regular __cdecl function.

This error message should not occur because it has been replaced with another one (E2475).

3.1.2.97 E2429: Not a valid partial specialization of 'specifier' (C++)

Internal compiler error.

3.1.2.98 E2430: Number of template parameters does not match in redeclaration of 'specifier' (C++)

If a template is redeclared with a different number of template parameters, this error will result. For example:

```
template<class T>
class foo;
template<class T, int U>
class foo;// error: parameter count mismatch!
```

3.1.2.99 E2477: Too few template parameters were declared for template 'template' (C++)

If a member declaration or definition occurs outside of a template class, and that outer declaration uses a different number of template parameters than the parent class, this error will result. For example:

```
template<class T, class U>
class foo {
void method();
};
template<class T>
void foo<T>::method() { }// error: too few template parameters!
```

3.1.2.100 E2478: Too many template parameters were declared for template 'template' (C++)

If a member declaration or definition occurs outside of a template class, and that outer declaration uses a different number of template parameters than the parent class, this error will result. For example:

```
template<class T, class U>
class foo {
void method();
};
template<class T, class U, class V>
void foo<T, U, V>::method() { }// error: too many parameters!
```

3.1.2.101 E2431: Non-type template parameters cannot be of floating point, class, or void type (C++)

Non-type template parameters are restricted as to what type they may be. Floating point, class and void types are illegal. For example:

```
template<float U>
class foo;// error: "U" cannot be of "float" type
```

3.1.2.102 E2434: Template declaration missing template parameters ('template<...>') (C++)

In a context where at least one template parameter is clearly required, if none are found this error will result. For example:

```
template<class T, template<> class U>
class foo;// error: template template parameters require
// at least one actual parameter to be declared
```

3.1.2.103 E2435: Too many template parameter sets were specified (C++)

If a member template is being defined outside of its parent class, and too many template parameter sets are declared, this error will result. For example:

```
template<class T>
class foo
{
template<class U>
void method(U a);
};
template<class T> template<class U> template<class V>
void foo<T>::method(U a);// error: too many parameter sets!
```

3.1.2.104 E2436: Default type for template template argument 'arg' does not name a primary template class (C++)

If a template template parameter is to have a default type, that type must either be a generic template class name, or another template template parameter.

```
template<class T>
class foo;
template<template<class T> class U = foo>
class bar;// OK: "foo" is a qualifying primary template
template<template<class T> class U = int>
class baz;// error: "int" is not a template class
```

3.1.2.105 E2437: 'typename' should be followed by a qualified, dependent type name (C++)

Whenever the "typename" keyword is used in a template declaration or definition, it should always name a dependent type. For example:

```
struct bar { };
template<class T>
class foo {
typename T::A *x;// OK: names a qualified type
typename T y;// error: not a qualified type
typename bar z;// error: not a dependent type
};
```

3.1.2.106 E2438: Template template arguments must name a class (C++)

A template template parameter must always declare a new class name. For example:

```
template<template<class T> int U>
class foo;// error: "U" is not a class tag name
template<template<class T> class V>
class bar;// OK: "V" is a class tag name
```

3.1.2.107 E2439: 'typename' is only allowed in template declarations (C++)

The "typename" keyword must only be used within template declarations and definitions.

3.1.2.108 E2440: Cannot generate specialization from 'specifier' because that type is not yet defined (C++)

The compiler no longer generates this error.

3.1.2.109 E2441: Instantiating 'specifier' (C++)

Whenever a compiler error occurs while instantiating a template type, the context of what was being instantiated at that point in time will be reported to the user, in order to aid in detection of the problem.

3.1.2.110 E2503: Missing or incorrect version of TypeLibImport.dll (C++)

This error occurs when the compiler is trying to access TypeLibImport.dll but it either can't find it, it was corrupted, or you have the wrong version of it installed on your computer. You can reinstall it from the product CD.

3.1.2.111 E2470: Need to include header <typeinfo> to use typeid (C++)

When you use the 'typeid' function, you have to include the <typeinfo> header, otherwise you will get syntax errors.

For example, consider a test case with the following code:

```
int func()
{
  char * name = typeid(int).name();  // This causes an error
}
```

3.1.2.112 E2514: Cannot (yet) use member overload resolution during template instantiation (C++)

You are trying to overload a member during template instantiation. You cannot have calls to overloaded constant functions within array bounds initializers, for example.

3.1.2.113 E2508: 'using' cannot refer to a template specialization (C++)

The `using` keyword cannot refer to a template specialization.

3.1.2.114 E2462: 'virtual' can only be used with non-template member functions (C++)

The 'virtual' keyword can only be applied to regular member functions, not to member template functions.

Consider a test case with the following code:

```
template <class T>
class myTemplateClass
{
  virtual int func1();                    // This is fine
  template <class T> virtual int func2(); // This causes an error
};
class myClass
{
  virtual int func1();                    // This is fine
  template <class T> virtual int func2(); // This causes an error
};
```

3.1.2.115 W8086: Incorrect use of #pragma alias "aliasName"="substituteName" (C++)

The directive #pragma alias is used to tell the linker that two identifier names are equivalent. You must put the two names in quotes.

You will receive this warning if you don't use pragma alias correctly. For example, the following two lines both generate this warning:

```
#pragma alias foo=bar
#pragma alias "foo" = bar
```

See the “#pragma alias” reference below for information on this pragma's usage.

See Also

#pragma (see page 696)

3.1.2.116 W8099: Static main is not treated as an entry point (C++)

The main function has been created as static, and as such cannot be used as a valid entry point.

Consider:

```
static void main(int argc, char**argv)
{
}
```

The above is valid C syntax, but cannot actually be called as the startup routine since it has been declared static.

3.1.2.117 W8093: Incorrect use of #pragma codeseg [seg_name] ["seg_class"] [group] (C++)

The #pragma codeseg directive can be used to set or reset the name, class, and group of a segment. You have to follow the exact syntax mentioned in the warning message, and all names are optional.

So these are all legal:

```
#pragma codeseg
#pragma codeseg foo
#pragma codeseg foo "bar"
#pragma codeseg foo "bar" foobar
```

But these are not:

```
#pragma codeseg ###
#pragma codeseg "foo" "bar"
#pragma codeseg foo "bar" foobar morefoobar
```

3.1.2.118 W8094: Incorrect use of #pragma comment(<type> [,"string"]) (C++)

The directive #pragma comment can be used to emit linker comment records.

In this message, <type> can be any of the following:

- user
- lib
- exestr
- linker

The type should be there but the string is optional.

3.1.2.119 W8085: Function 'function' redefined as non-inline (C++)

This warning is used to indicate when a certain function, which has been declared inline in one location, is redefined in another location to be non-inline.

3.1.2.120 W8105: %s member '%s' in class without constructors (C++)

A class that contains constant or reference members (or both) must have at least one user-defined constructor.

Otherwise, there would be no way to ever initialize such members.

3.1.2.121 W8095: Incorrect use of #pragma message("string") (C++)

You can use pragma message to emit a message to the command line or to the message window. You would get this warning if you use the incorrect syntax, so

```
#pragma message( "hi there" )
```

is correct, but the following code would generate the warning:

```
#pragma message( badly formed )
```

3.1.2.122 W8098: Multi-character character constant (C++)

This warning is issued when the compiler detects a multi-character integer constant, such as:

```
int foo = 'abcd';
```

The problem with this construct is that the byte order of the characters is implementation dependent.

3.1.2.123 W8096: Incorrect use of #pragma code_seg(["seg_name"[,"seg_class"]]) (C++)

Pragma code_seg is similar to pragma codeseg, but with this one you can only modify the name and the class of a code segment. If you use the wrong syntax, you get this warning.

The following examples show the correct usage:

```
#pragma code_seg()
#pragma code_seg("foo")
#pragma code_seg("foo", "bar")
```

However, the following incorrect examples will all produce the warning:

```
#pragma code_seg(foo)
#pragma code_seg(foo, bar)
```

3.1.2.124 W8083: Pragma pack pop with no matching pack push (C++)

Each #pragma pack(pop) should have a matching preceding #pragma pack(push) in the same translation unit. Pairs of 'push' and 'pop' can be nested.

For example:

```
#pragma pack( push )
#pragma pack( push )
#pragma pack( pop )
#pragma pack( pop )
#pragma pack( pop )   // This causes an error
```

3.1.2.125 W8097: Not all options can be restored at this time (C++)

Your program has a #pragma pop at a place where it can't restore options.

For example:

```
#pragma option push -v
int main()
{
  int i;
  i = 1;
#pragma option pop
  return i;
}
```

For this example, compile with -v-. The message happens because the first #pragma causes debug info to change state (turns it on). Then, in the middle of the function where it is useless to toggle the debug info state, the #pragma pop attempts to return to the former state.

3.1.2.126 W8084: Suggest parentheses to clarify precedence (C++)

This warning indicates that several operators used in one expression might cause confusion about the applicable operator precedence rules. The warning helps create code that is more easy to understand and potentially less ambiguous.

For example, compile the following code using the `-w` command line option:

```
int j, k, l;
int main()
{
  return j < k & l;  // This causes an error
}
//
```

3.1.2.127 W8092: 'type' argument 'specifier' passed to 'function' is not an iterator: 'type' iterator required (C++)

An argument that is not an iterator is being used with an STL algorithm that requires an iterator.

3.1.2.128 W8087: 'operator::operator==' must be publicly visible to be contained by a 'type' (C++)

A type that is being used with an STL container has a private 'operator==='.

3.1.2.129 W8090: 'type::operator<' must be publicly visible to be used with 'type' (C++)

A type that is being used with an STL container has a private 'operator<'. The type you're trying to use must be made public.

3.1.2.130 W8089: 'type::operator<' must be publicly visible to be contained by a 'type' (C++)

The type that is being used for an STL container has a private 'operator<'. The type that is being contained (type::operator) must be a public type.

For example, if you were trying to instantiate a class type "vector<blah>", the error would be:

```
'blah::operator<' must be publicly visible to be contained by a 'vector'
```

3.1.2.131 W8091: 'type' argument 'specifier' passed to 'function' is a 'iterator category' iterator: 'iterator category' iterator required (C++)

An incorrect iterator category is being used with an STL algorithm.

3.1.2.132 W8076: Template instance 'specifier' is already instantiated (C++)

You are trying to explicitly instantiate a template that was already implicitly instantiated.

If –A is not enabled and an attempt is made to explicitly instantiate a specialization which has already been either implicitly or explicitly instantiated, this error will result.

3.1.2.133 W8077: Explicitly specializing an explicitly specialized class member makes no sense (C++)

Internal error. This warning is no longer generated by the compiler.

The following code is illegal:

```
template<class T> class foo { int x; }
template<> class foo<int> { int y; }
template<> int foo<int>::y;   // error: cannot explicitly specialize of 'foo<int>::x'
```

3.1.2.134 Informational messages (C++)

The compiler displays status information while compiling if you have checked "Show general messages" on the Compiler page of the Project Options dialog box. Most of the messages are self-explanatory and state information about compiling and linking; for example:

```
[C++] Compiling: D:\Program Files\Borland\CBuilder\Bin\Unit1.cpp
```

You may also see a message such as

```
[C++] Including clx.h instead of clx.h due to -Hr switch
```

This message indicates that the IDE-managed precompiled header file was replaced due to additional CLX classes being included in the project. It does not indicate a failure condition. Sometimes the header file name being included may be the same if it is being included with a different #define statement.

3.1.2.135 E2196: Cannot take address of member function 'function' (C++)

An expression takes the address of a class member function, but this member function was not found in the program being debugged. The evaluator issues this message.

3.1.2.136 F1002: Unable to create output file 'filename' (C++)

This error occurs if the work disk is full or write protected.

This error also occurs if the output directory does not exist.

Solutions

If the disk is full, try deleting unneeded files and restarting the compilation.

If the disk is write-protected, move the source files to a writeable disk and restart the compilation.

3.1.2.137 F1003: Error directive: 'message' (C++)

This message is issued when an `#error` directive is processed in the source file.

'message' is the text of the `#error` directive.

3.1.2.138 F1004: Internal compiler error (C++)

An error occurred in the internal logic of the compiler. This error shouldn't occur in practice, but is generated in the event that a more specific error message is not available.

3.1.2.139 F1006: Bad call of intrinsic function (C++)

You have used an intrinsic function without supplying a prototype. You may have supplied a prototype for an intrinsic function that was not what the compiler expected.

3.1.2.140 F1007: Irreducible expression tree (C++)

An expression on the indicated line of the source file caused the code generator to be unable to generate code. Avoid using the expression. Notify CodeGear if an expression consistently reproduces this error.

3.1.2.141 F1009: Unable to open input file 'filename' (C++)

This error occurs if the source file can't be found.

Check the spelling of the name. Make sure the file is on the specified disk or directory.

Verify that the proper directory paths are listed. If multiple paths are required, use a semicolon to separate them.

3.1.2.142 F1011: Register allocation failure (C++)

Possible Causes

An expression on the indicated line of the source file was so complicated that the code generator could not generate code for it.

Solutions

Simplify the expression. If this does not solve the problem, avoid the expression.

Notify CodeGear if an expression can consistently reproduce this error.

3.1.2.143 F1012: Compiler stack overflow (C++)

The compiler's stack has overflowed. This can be caused by a number of things, among them deeply nested statements in a function body (for example, if/else) or expressions with a large number of operands. You must simplify your code if this message occurs. Adding more memory to your system will not help.

3.1.2.144 F1013: Error writing output file (C++)

A DOS error that prevents the C++ IDE from writing an .OBJ, .EXE, or temporary file.

Solutions

Make sure that the Output directory in the Directories dialog box is a valid directory.

Check that there is enough free disk space.

3.1.2.145 F1000: Compiler table limit exceeded (C++)

One of the compiler's internal tables overflowed.

This usually means that the module being compiled contains too many function bodies.

This limitation will not be solved by making more memory available to the compiler. You need to simplify the file being compiled.

3.1.2.146 F1005: Include files nested too deep (C++)

This message flags (directly or indirectly) recursive `#include` directives.

3.1.2.147 F1008: Out of memory (C++)

The total working storage is exhausted.

This error can occur in the following circumstances:

- Not enough virtual memory is available for compiling a particular file. In this case, shut down any other concurrent applications. You may also try to reconfigure your machine for more available virtual memory, or break up the source file being compiled into smaller separate components. You can also compile the file on a system with more available RAM.
- The compiler has encountered an exceedingly complex or long expression at the line indicated and has insufficient reserves to parse it. Break the expression down into separate statements.

3.1.2.148 F1010: Unable to open 'filename' (C++)

This error occurs if the specified file can't be opened.

Make sure the file is on the specified disk or directory. Verify the proper paths are listed. If multiple paths are required, use a semicolon to separate them.

3.1.2.149 E2000: 286/287 instructions not enabled (C++)

Use the -2 command-line compiler option to enable 286/287 opcodes. Be aware that the resulting code cannot be run on 8086- and 8088-based machines.

3.1.2.150 Abnormal program termination (C++)

The program called abort because there wasn't enough memory to execute.

This message can be caused by memory overwrites.

3.1.2.151 E2009: Attempt to grant or reduce access to 'identifier' (C++)

A C++ derived class can modify the access rights of a base class member, but only by restoring it to the rights in the base class.

It can't add or reduce access rights.

3.1.2.152 E2011: Illegal to take address of bit field (C++)

It is not legal to take the address of a bit field, although you can take the address of other kinds of fields.

3.1.2.153 E2010: Cannot add or subtract relocatable symbols (C++)

The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant.

Variables, procedures, functions, and labels are relocatable symbols.

3.1.2.154 E2013: 'function1' cannot be distinguished from 'function2' (C++)

The parameter type lists in the declarations of these two functions do not differ enough to tell them apart.

Try changing the order of parameters or the type of a parameter in one declaration.

3.1.2.155 E2014: Member is ambiguous: 'member1' and 'member2' (C++)

You must qualify the member reference with the appropriate base class name.

In C++ class 'class', member 'member' can be found in more than one base class, and it was not qualified to indicate which one you meant.

This applies only in multiple inheritance, where the member name in each base class is not hidden by the same member name in a derived class on the same path.

The C++ language rules require that this test for ambiguity be made before checking for access rights (private, protected, public).

It is possible to get this message even though only one (or none) of the members can be accessed.

3.1.2.156 E2015: Ambiguity between 'function1' and 'function2' (C++)

Both of the named overloaded functions could be used with the supplied parameters.

This ambiguity is not allowed.

3.1.2.157 E2017: Ambiguous member name 'name' (C++)

Whenever a structure member name is used in inline assembly, such a name must be unique. (If it is defined in more than one structure, all of the definitions must agree as to its type and offset within the structures). In this case, an ambiguous member name has been used.

For example:

```
struct A
{
   int a;
   int b;
};
...
asm ax,.a;
```

3.1.2.158 E2019: 'identifier' cannot be declared in an anonymous union (C++)

The compiler found a declaration for a member function or static member in an anonymous union.

Such unions can only contain data members.

3.1.2.159 E2020: Global anonymous union not static (C++)

In C++, a global anonymous union at the file level must be static.

3.1.2.160 E2022: Array size too large (C++)

The declared array is larger than 64K and the 'huge' keyword was not used.

If you need an array of this size, either use the 'huge' modifier, like this:

```
int huge array[70000L];  /* Allocate 140000 bytes */
```

or dynamically allocate it with farmalloc( ) or farcalloc( ), like this:

```
int huge *array = (int huge *) farmalloc (sizeof (int) * 70000); ?? Allocate 140,000 bytes
```

3.1.2.161 E2024: Cannot modify a const object (C++)

This indicates an illegal operation on an object declared to be const, such as an assignment to the object.

3.1.2.162 E2025: Assignment to 'this' not allowed, use X::operator new instead (C++)

In early versions of C++, the only way to control allocation of class of objects was by assigning to the 'this' parameter inside a constructor.

This practice is no longer allowed, because a better, safer, and more general technique is to define a member function operator new instead.

For example:

```
this = malloc(n);
```

3.1.2.163 E2026: Assembler statement too long (C++)

Inline assembly statements can't be longer than 480 bytes.

3.1.2.164 E2001: Constructors and destructors not allowed in __automated section (C++)

Only member function declarations are allowed in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int__fastcall fooInt(int);// OK
clxclass()  {}// Error
};
```

3.1.2.165 E2002: Only __fastcall functions allowed in __automated section (C++)

The calling convention for functions declared in an __automated section must be __fastcall.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int__fastcall fooInt(int);// OK
int__cdecl    barInt(int);// Error
};
```

3.1.2.166 E2003: Data member definition not allowed in __automated section (C++)

Only member function declarations are allowed in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int__fastcall fooInt(int);// OK
intmemInt;// Error
};
```

3.1.2.167 E2004: Only read or write clause allowed in property declaration in __automated section (C++)

Storage specifiers stored, default, and nodefault are not allowed in property declarations in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
int __fastcall Get(void);
__automated:
int __property ip1 = { read = Get };      // OK
int __property ip2 = { read = Get, default = 42 }; // Error
};
```

3.1.2.168 E2005: Redeclaration of property not allowed in __automated section (C++)

If you declare a property in an __automated section it has be a new declaration. Property hoisting is not allowed.

Example

```
struct__declspec(delphiclass) clxbaseclass
{
int __fastcall Get(void);
void __fastcall Set(int);
int __property ip1 = { read = Get, write = Set };
};
structclxderivedclass : clxbaseclass
{
int __fastcall NewGetter(void);
__automated:
__property ip1;// Error
int __property ip2 = { read = Get, write = Set }; // OK
};
```

3.1.2.169 E2027: Must take address of a memory location (C++)

Your source file used the address-of operator (&) with an expression that can't be used that way; for example, a register variable.

3.1.2.170 E2028: operator -> must return a pointer or a class (C++)

The C++ operator -> function must be declared to either return a class or a pointer to a class (or struct or union).

In either case, it must be something to which the -> operator can be applied.

3.1.2.171 E2029: 'identifier' must be a previously defined class or struct (C++)

You are attempting to declare 'identifier' to be a base class, but either it is not a class or it has not yet been fully defined.

Correct the name or rearrange the declarations.

3.1.2.172 E2030: Misplaced break (C++)

The compiler encountered a break statement outside a switch or looping construct.

You can only use break statements inside of switch statements or loops.

3.1.2.173 E2031: Cannot cast from 'type1' to 'type2' (C++)

A cast from type 'ident1' to type 'ident2' is not allowed.

In C++, you cannot cast a member function pointer to a normal function pointer.

For example:

```
class A {
public:
   int myex();
};
typedef int (*fp)();
test()
{
   fp myfp = (fp) &A::myex; //error
   return myfp();
}
```

The reason being that a class member function takes a hidden parameter, the this pointer, thus it behaves very differently than a normal function pointer.

A static member function behaves as normal function pointer and can be cast.

For example:

```
class A {
public:
   static int myex();
};
typedef int (*fp)();
test()
{
   fp myfp = (fp) &A::myex; //ok
   return myfp();
}
```

However, static member functions can only access static data members of the class.

In C

- A pointer can be cast to an integral type or to another pointer.
- An integral type can be cast to any integral, floating, or pointer type.
- A floating type can be cast to an integral or floating type.

Structures and arrays can't be cast to or from.

You usually can't cast from a void type.

In C++

User-defined conversions and constructors are checked for. If one can't be found, the preceding rules apply (except for pointers to class members).

Among integral types, only a constant zero can be cast to a member pointer.

A member pointer can be cast to an integral type or to a similar member pointer.

A similar member pointer points to a data member (or to a function) if the original does. The qualifying class of the type being cast to must be the same as (or a base class of) the original.

3.1.2.174 E2033: Misplaced continue (C++)

The compiler encountered a continue statement outside a looping construct.

3.1.2.175 E2034: Cannot convert 'type1' to 'type2' (C++)

An assignment, initialization, or expression requires the specified type conversion to be performed, but the conversion is not

legal.

In C++, the compiler will convert one function pointer to another only if the signature for the functions are the same. Signature refers to the arguments and return type of the function. For example:

```
myex( int );
typedef int ( *ffp )( float );
test()
{
   ffp fp = myex; //error
}
```

Seeing that myex takes an int for its argument, and fp is a pointer to a function which takes a float as argument, the compiler will not convert it for you.

In cases where this is what is intended, performing a typecast is necessary:

```
myex( int );
typedef int ( *ffp )( float );
test()
{
   ffp fp = (ffp)myex;  //ok
}
```

3.1.2.176 E2036: Conversion operator cannot have a return type specification (C++)

This C++ type conversion member function specifies a return type different from the type itself.

A declaration for conversion function operator can't specify any return type.

3.1.2.177 E2037: The constructor 'constructor' is not allowed (C++)

Constructors of the form

```
  X(X);  //{ Error
};
```

are not allowed.

This is the correct way to write a copy constructor:

```
class X {
  X(const X&);// Copy constructor
};
```

3.1.2.178 E2039: Misplaced decimal point (C++)

The compiler encountered a decimal point in a floating-point constant as part of the exponent.

3.1.2.179 E2041: Incorrect use of default (C++)

The compiler found no colon after the default keyword.

3.1.2.180 E2042: Declare operator delete (void*) or (void*, size_t) (C++)

E2043 Declare operator delete[] (void) or (void*, size_t)*Compiler error

Declare the operator delete with one of the following:

1.A single void* parameter, or

2.A second parameter of type size_t

If you use the second version, it will be used in preference to the first version.

The global operator delete can only be declared using the single-parameter form.

3.1.2.181 E2044: operator delete must return void (C++)

*E2044 operator delete[] must return void*Compiler error

This C++ overloaded operator delete was declared in some other way.

Declare the operator delete with one of the following:

1.A single void* parameter, or

2.A second parameter of type size_t

If you use the second version, it will be used in preference to the first version.

The global operator delete can only be declared using the single-parameter form.

3.1.2.182 E2045: Destructor name must match the class name (C++)

In a C++ class, the tilde (~) introduces a declaration for the class destructor.

The name of the destructor must be same as the class name.

In your source file, the ~ preceded some other name.

3.1.2.183 E2048: Unknown preprocessor directive: 'identifier' (C++)

The compiler encountered a # character at the beginning of a line. The directive name that followed the # was not one of the following:

- define (see page 690)
- else (see page 688)
- endif
- if
- ifdef (see page 688)
- ifndef (see page 688)
- include (see page 692)

- line (see page 693)
- pragma
- undef (see page 692)

3.1.2.184 E2046: Bad file name format in include directive OR Bad file name format in line directive (C++)

Include and line directive file names must be surrounded by quotes ("filename.h") or angle brackets (<filename.h>).

The file name was missing the opening quote or angle bracket.

If a macro was used, the resulting expansion text is not surrounded by quote marks.

3.1.2.185 E2051: Invalid use of dot (C++)

An identifier must immediately follow a period operator (.). This is a rare message that can only occur in some specialized inline assembly statements.

Example

```
struct foo {
  int x;
  int y;
}p = {0,0};
int y;
int main (void)
{
  asm mov eax.(foo)x, 1;
  asm mov eax.(foo)4, 1;       /* Error: Invalid use of dot */
  return 0;
}
```

3.1.2.186 E2053: Misplaced elif directive (C++)

The compiler encountered an `#elif` directive without any matching `#if`, `#ifdef`, or `#ifndef` directive.

3.1.2.187 E2054: Misplaced else (C++)

The compiler encountered an `else` statement without a matching if statement.

Possible Causes

- An extra "else" statement
- An extra semicolon
- Missing braces
- Some syntax error in a previous "if" statement

3.1.2.188 E2055: Misplaced else directive (C++)

The compiler encountered an `#else` directive without any matching `#if`, `#ifdef`, or `#ifndef` directive.

3.1.2.189 E2056: Misplaced endif directive (C++)

The compiler encountered an `#endif` directive without any matching `#if`, `#ifdef`, or `#ifndef` directive.

3.1.2.190 E2059: Unknown language, must be C or C++ (C++)

In the C++ construction

```
extern "name" type func( /*...*/ );
```

the given "name" must be "C" or "C++" (use the quotes); other language names are not recognized.

You can declare an external Pascal function without the compiler's renaming like this:

```
extern "C" int pascal func( /*...*/ );
```

To declare a (possibly overloaded) C++ function as Pascal and allow the usual compiler renaming (to allow overloading), you can do this:

```
extern int pascal func( /*...*/ );
```

3.1.2.191 E2060: Illegal use of floating point (C++)

Floating-point operands are not allowed in these operators

- shift (SHL, SHR)
- bitwise Boolean (AND, OR, XOR, NOT)
- conditional (? :)
- indirection (*)
- certain others

The compiler found a floating-point operand with one of these prohibited operators.

3.1.2.192 E2061: Friends must be functions or classes (C++)

A friend of a C++ class must be a function or another class.

3.1.2.193 E2062: Invalid indirection (C++)

The indirection operator (*) requires a pointer as the operand.

Example

```
int main (void)
{
  int p;
  *p = 10;     /* ERROR: Invalid Indirection */
  return 0;
}
```

3.1.2.194 E2063: Illegal initialization (C++)

Initializations must be one of the following:

- constant expressions
- the address of a global extern or static variable plus or minus a constant

3.1.2.195 E2064: Cannot initialize 'type1' with 'type2' (C++)

You are attempting to initialize an object of type 'type1' with a value of type 'type2' which is not allowed.

The rules for initialization are essentially the same as for assignment.

3.1.2.196 E2068: 'identifier' is not a non-static data member and can't be initialized here (C++)

Only data members can be initialized in the initializers of a constructor.

This message means that the list includes a static member or function member.

Static members must be initialized outside of the class, for example:

```
class A { static int i; };
int A::i = -1;
```

3.1.2.197 E2069: Illegal use of member pointer (C++)

Pointers to class members can only be passed as arguments to functions, or used with the following operators:

- assignment operators
- comparison operators
- `.*`
- `–>*`
- `?:` conditional (ternary) operator
- && logical AND (see page 531) operator
- || logical OR (see page 559) operator

The compiler has encountered a member pointer being used with a different operator.

In order to call a member function pointer, one must supply an instance of the class for it to call upon.

For example:

```
class A {
public:
   myex();
};
typedef int (A::*Amfptr)();
myex()
{
   Amfptr mmyex = &A::myex;
   return (*mmyex)();  //error
}
```

This will compile:

```
class A {
public:
   myex();
};
typedef int (A::*Amfptr)();
foo()
{
   A a;
   Amfptr mmyex = &A::myex;
   return (a.*mmyex)();
}
```

3.1.2.198 E2071: operator new must have an initial parameter of type size_t (C++)

*E2071 Operator new[] must have an initial parameter of type size_t*Compiler error

Operator new can be declared with an arbitrary number of parameters.

It must always have at least one, the amount of space to allocate.

3.1.2.199 E2072: Operator new[] must return an object of type void (C++)

This C++ overloaded operator new was declared in some other way.

3.1.2.200 E2075: Incorrect 'type' option: option (C++)

An error has occurred in either the configuration file or a command-line option. The compiler may not have recognized the configuration file parameter as legal; check for a preceding hyphen (-), or the compiler may not have recognized the command-line parameter as legal.

This error can also occur if you use a `#pragma` option in your code with an invalid option.

3.1.2.201 E2076: Overloadable operator expected (C++)

Almost all C++ operators can be overloaded.

These are the only ones that can't be overloaded:

- the field-selection dot (`.`)
- dot-star (`.*`)
- double colon (`::`)
- conditional expression (`?:`)

The preprocessor operators (# and ##) are not C or C++ language operators and thus can't be overloaded.

Other non-operator punctuation, such as semicolon (;), can't be overloaded.

3.1.2.202 E2080: 'function' must be declared with one parameter (C++)

This C++ operator function was incorrectly declared with more than one parameter.

3.1.2.203 E2077: 'operator' must be declared with one or no parameters (C++)

When operator ++ or operator -- is declared as a member function, it must be declared to take either:

- No parameters (for the prefix version of the operator), or
- One parameter of type int (for the postfix version)

3.1.2.204 E2079: 'function' must be declared with no parameters (C++)

This C++ operator function was incorrectly declared with parameters.

3.1.2.205 E2078: 'operator' must be declared with one or two parameters (C++)

When operator ++ or operator -- is declared as a non-member function, it must be declared to take either:

- one parameter (for the prefix version of the operator), or
- two parameters (for the postfix version)

3.1.2.206 E2081: 'function' must be declared with two parameters (C++)

This C++ operator function was incorrectly declared with other than two parameters.

3.1.2.207 E2082: 'identifier' must be a member function or have a parameter of class type (C++)

Most C++ operator functions must have an implicit or explicit parameter of class type.

This operator function was declared outside a class and does not have an explicit parameter of class type.

3.1.2.208 E2083: Last parameter of 'operator' must have type 'int' (C++)

When a postfix operator ++ or operator -- is overloaded, the last parameter must be declared with the type int.

3.1.2.209 E2084: Parameter names are used only with a function body (C++)

When declaring a function (not defining it with a function body), you must use either empty parentheses or a function prototype.

A list of parameter names only is not allowed.

Example declarations

```
int func();             /* declaration without prototype -- OK */
int func(int, int);     /* declaration with prototype -- OK */
int func(int i, int j); /* parameter names in prototype -- OK */
int func(i, j);         /* parameter names only -- ILLEGAL */
```

3.1.2.210 E2085: Invalid pointer addition (C++)

Your source file attempted to add two pointers together.

3.1.2.211 E2086: Illegal pointer subtraction (C++)

This is caused by attempting to subtract a pointer from a non-pointer.

3.1.2.212 E2087: Illegal use of pointer (C++)

Pointers can only be used with these operators:

- addition(+)
- subtraction(-)
- assignment(=)
- comparison(==)
- indirection(*)
- arrow(->)

Your source file used a pointer with some other operator.

Example

```
int main (void)
{
  char *p;
  p /= 7;     /* ERROR: Illegal Use of Pointer */
  return 0;
}
```

3.1.2.213 E2088: Bad syntax for pure function definition (C++)

Pure virtual functions are specified by appending "= 0" to the declaration, like this:

```
class A { virtual void f () = 0;}
class B : public A { void f () {}; }
```

You wrote something similar, but it was not correct.

3.1.2.214 E2089: Identifier 'identifier' cannot have a type qualifier (C++)

A C++ qualifier class::identifier can't be applied here.

A qualifier is not allowed on the following:

- typedef names
- function declarations (except definitions at the file level)
- on local variables or parameters of functions
- on a class member--except to use its own class as a qualifier (redundant but legal).

3.1.2.215 E2090: Qualifier 'identifier' is not a class or namespace name (C++)

The C++ qualifier in the construction qual::identifier is not the name of a struct or class.

3.1.2.216 E2092: Storage class 'storage class' is not allowed here (C++)

The given storage class is not allowed here.

Probably two storage classes were specified, and only one can be given.

3.1.2.217 E2096: Illegal structure operation (C++)

Structures can only be used with dot (.), address-of (&) or assignment (=) operators, or be passed to or from a function as parameters.

The compiler encountered a structure being used with some other operator.

3.1.2.218 E2104: Invalid use of template keyword (C++)

You can only use a template class name without specifying its actual arguments inside a template definition.

Using a template class name without specifying its actual arguments outside a template definition is illegal.

3.1.2.219 E2108: Improper use of typedef 'identifier' (C++)

Your source file used a typedef symbol where a variable should appear in an expression.

Check for the declaration of the symbol and possible misspellings.

3.1.2.220 E2109: Not an allowed type (C++)

Your source file declared some sort of forbidden type; for example, a function returning a function or array.

3.1.2.221 E2110: Incompatible type conversion (C++)

The cast requested can't be done.

3.1.2.222 E2113: Virtual function 'function1' conflicts with base class 'base' (C++)

A virtual function has the same argument types as one in a base class, but differs in one or more of the following:

- Return type
- Calling convention
- Exception specification (throw list)

3.1.2.223 E2114: Multiple base classes require explicit class names (C++)

In a C++ class constructor, if there is more than one immediate base class, each base class constructor call in the constructor header must include the base class name.

3.1.2.224 E2115: Bit field too large (C++)

This error occurs when you supply a bit field with more than 16 bits.

3.1.2.225 E2116: Bit fields must contain at least one bit (C++)

You can't declare a named bit field to have 0 (or less than 0) bits.

You can declare an unnamed bit field to have 0 bits.

This is a convention used to force alignment of the following bit field to a byte boundary (or to a word boundary.

3.1.2.226 W8005: Bit fields must be signed or unsigned int (C++)

In ANSI C, bit fields may only be signed or unsigned int (not char or long, for example).

3.1.2.227 E2119: User break (C++)

You typed a Ctrl+Break while compiling in the IDE.

(This is not an error, just a confirmation.)

3.1.2.228 E2111: Type 'typename' may not be defined here (C++)

Class and enumeration types may not be defined in a function return type, a function argument type, a conversion operator type, or the type specified in a cast.

You must define the given type before using it in one of these contexts.

`Note:`This error message is often the result of a missing semicolon (`;`) for a class declaration. You might want to verify that all the class declarations preceding the line on which the error occurred end with a semicolon.

3.1.2.229 E2121: Function call missing) (C++)

The function call argument list had some sort of syntax error, such as a missing or mismatched right parenthesis.

3.1.2.230 E2123: Class 'class' may not contain pure functions (C++)

The class being declared cannot be abstract, and therefore it cannot contain any pure functions.

3.1.2.231 E2126: Case bypasses initialization of a local variable (C++)

In C++ it is illegal to bypass the initialization of a local variable.

This error indicates a case label that can transfer control past this local variable.

3.1.2.232 E2127: Case statement missing : (C++)

A case statement must have a constant expression followed by a colon.

The expression in the case statement either was missing a colon or had an extra symbol before the colon.

3.1.2.233 E2128: Case outside of switch (C++)

The compiler encountered a case statement outside a switch statement.

This is often caused by mismatched braces.

3.1.2.234 E2129: Character constant too long (or empty) (C++)

Character constants can only be one or two characters long.

3.1.2.235 E2133: Unable to execute command 'command' (C++)

The linker or assembler cannot be found, or possibly the disk is bad.

3.1.2.236 E2134: Compound statement missing closing brace (C++)

The compiler reached the end of the source file and found no closing brace.

This is most commonly caused by mismatched braces.

3.1.2.237 E2137: Destructor for 'class' required in conditional expression (C++)

If the compiler must create a temporary local variable in a conditional expression, it has no good place to call the destructor because the variable might or might not have been initialized.

The temporary can be explicitly created, as with classname(val, val), or implicitly created by some other code.

You should recast your code to eliminate this temporary value.

3.1.2.238 E2135: Constructor/Destructor cannot be declared 'const' or 'volatile' (C++)

A constructor or destructor has been declared as `const` or `volatile`.

This is not allowed.

3.1.2.239 E2138: Conflicting type modifiers (C++)

This occurs when a declaration is given that includes more than one addressing modifier on a pointer or more than one language modifier for a function.

Only one language modifier (for example, `__cdecl`, `__pascal`, or `__fastcall`) can be given for a function.

3.1.2.240 E2136: Constructor cannot have a return type specification (C++)

C++ constructors have an implicit return type used by the compiler, but you can't declare a return type or return a value.

3.1.2.241 E2038: Cannot declare or define 'identifier' here: wrong namespace (C++)

You tried to declare a template in an illegal place or a namespace member outside of its namespace.

3.1.2.242 E2154: Cannot define 'identifier' using a namespace alias (C++)

You cannot use a namespace alias to define a namespace member outside of its namespace.

3.1.2.243 E2421: Cannot use local type 'identifier' as template argument (C++)

A local type was used in an actual template type argument, which is illegal.

3.1.2.244 E2035: Conversions of class to itself or base class not allowed (C++)

You tried to define a conversion operator to the same class or a base class.

3.1.2.245 E2139: Declaration missing ; (C++)

Your source file contained a struct or union field declaration that was not followed by a semicolon.

Check previous lines for a missing semicolon.

3.1.2.246 E2140: Declaration is not allowed here (C++)

Declarations can't be used as the control statement for while, for, do, if, or switch statements.

3.1.2.247 E2141: Declaration syntax error (C++)

Your source file contained a declaration that was missing a symbol or had an extra symbol added to it.

Check for a missing semicolon or parenthesis on that line or on previous lines.

3.1.2.248 E2142: Base class 'class' contains dynamically dispatchable functions (C++)

This error occurs when a class containing a DDVT function attempts to inherit DDVT functions from multiple parent classes.

Currently, dynamically dispatched virtual tables do not support the use of multiple inheritance.

3.1.2.249 E2143: Matching base class function 'function' has different dispatch number (C++)

If a DDVT function is declared in a derived class, the matching base class function must have the same dispatch number as the derived function.

3.1.2.250 E2144: Matching base class function 'function' is not dynamic (C++)

If a DDVT function is declared in a derived class, the matching base class function must also be dynamic.

3.1.2.251 E2145: Functions 'function1' and 'function2' both use the same dispatch number (C++)

This error indicates a dynamically dispatched virtual table (DDVT) problem.

3.1.2.252 E2146: Need an identifier to declare (C++)

In this context, an identifier was expected to complete the declaration.

This might be a typedef with no name, or an extra semicolon at file level.

In C++, it might be a class name improperly used as another kind of identifier.

3.1.2.253 E2147: 'identifier' cannot start a parameter declaration (C++)

An undefined 'identifier' was found at the start of an argument in a function declarator.

Often the type name is misspelled or the type declaration is missing. This is usually caused by not including the appropriate header file.

3.1.2.254 E2150: Type mismatch in default argument value (C++)

The default parameter value given could not be converted to the type of the parameter.

The message "Type mismatch in default argument value" is used when the parameter was not given a name.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is most often "Cannot convert 'type1' to 'type2'" but the mismatch could be due to another reason.

3.1.2.255 E2152: Default expression may not use local variables (C++)

A default argument expression is not allowed to use any local variables or other parameters.

3.1.2.256 E2153: Define directive needs an identifier (C++)

The first non-whitespace character after a `#define` must be an identifier.

The compiler found some other character.

3.1.2.257 E2155: Too many default cases (C++)

The compiler encountered more than one default statement in a single switch.

3.1.2.258 E2156: Default outside of switch (C++)

The compiler encountered a default statement outside a switch statement.

This is most commonly caused by mismatched braces.

3.1.2.259 E2158: Operand of 'delete' must be non-const pointer (C++)

It is illegal to delete a variable that is not a pointer. It is also illegal to delete a pointer to a constant.

For example:

```
const int x=10;
   const int * a = &x;
   int * const b = new int;
   int &c = *b;
   delete a;   //illegal - deleting pointer to constant
   delete b;   //legal
   delete c;   //illegal - operand not of pointer type
               //should use 'delete&c' instead
```

3.1.2.260 E2159: Trying to derive a far class from the huge base 'base' (C++)

This error is no longer generated by the compiler.

3.1.2.261 E2160: Trying to derive a far class from the near base 'base' (C++)

If a class is declared (or defaults to) near, all derived classes must also be near.

3.1.2.262 E2161: Trying to derive a huge class from the far base 'base' (C++)

This error is no longer generated by the compiler.

3.1.2.263 E2162: Trying to derive a huge class from the near base 'base' (C++)

This error is no longer generated by the compiler.

3.1.2.264 E2163: Trying to derive a near class from the far base 'base' (C++)

If a class is declared (or defaults to) far, all derived classes must also be far.

3.1.2.265 E2164: Trying to derive a near class from the huge base 'base' (C++)

This error is no longer generated by the compiler.

3.1.2.266 E2165: Destructor cannot have a return type specification (C++)

C++ destructors never return a value, and you can't declare a return type or return a value.

3.1.2.267 E2166: Destructor for 'class' is not accessible (C++)

The destructor for this C++ class is protected or private, and can't be accessed here to destroy the class.

If a class destructor is private, the class can't be destroyed, and thus can never be used. This is probably an error.

A protected destructor can be accessed only from derived classes.

This is a useful way to ensure that no instance of a base class is ever created, but only classes derived from it.

3.1.2.268 E2167: 'function' was previously declared with the language 'language' (C++)

Only one language modifier (cdecl pascal) can be given for a function.

This function has been declared with different language modifiers in two locations.

3.1.2.269 E2168: Division by zero (C++)

Your source file contains a divide or remainder in a constant expression with a zero divisor.

3.1.2.270 E2169: 'identifier' specifies multiple or duplicate access (C++)

A base class can be declared public or private, but not both.

This access specifier can appear no more than once for a base class.

3.1.2.271 E2170: Base class 'class' is included more than once (C++)

A C++ class can be derived from any number of base classes, but can be directly derived from a given class only once.

3.1.2.272 E2171: Body has already been defined for function 'function' (C++)

A function with this name and type was previously supplied a function body.

A function body can only be supplied once.

One cause of this error is not declaring a default constructor which you implement. For example:

```
class A {
public:
   virtual myex();
};
A::A() {} // error
```

Having not seen you declare the default constructor in the class declaration, the compiler has had to generate one, thus giving the error message when it sees one. this is a correct example:

```
class A {
public:
   A();
   virtual myex();
};
A::A() {}
```

3.1.2.273 E2172: Duplicate case (C++)

Each case of a switch statement must have a unique constant expression value.

3.1.2.274 E2175: Too many storage classes in declaration (C++)

A declaration can never have more than one storage class, either Auto, Register, Static, or Extern.

3.1.2.275 E2176: Too many types in declaration (C++)

A declaration can never have more than one basic type. Examples of basic types are:

- char
- class
- int
- float
- double
- struct
- union
- enum
- typedef name

3.1.2.276 E2179: virtual specified more than once (C++)

The C++ reserved word "virtual" can appear only once in one member function declaration.

3.1.2.277 E2007: Dispid only allowed in __automated sections (C++)

The definition of dispids is only permitted in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
int __fastcall foo1(void) __dispid(42);// Error
__automated:
int __fastcall foo2(void) __dispid(43);// OK
};
```

3.1.2.278 Divide error (C++)

You tried to divide an integer by zero, which is illegal.

3.1.2.279 E2182: Illegal parameter to __emit__ (C++)

There are some restrictions on inserting literal values directly into your code with the __emit__ function.

For example, you cannot give a local variable as a parameter to __emit__.

3.1.2.280 E2183: File must contain at least one external declaration (C++)

This compilation unit was logically empty, containing no external declarations.

ANSI C and C++ require that something be declared in the compilation unit.

3.1.2.281 E2184: Enum syntax error (C++)

An `enum` declaration did not contain a properly formed list of identifiers.

3.1.2.282 E2185: The value for 'identifier' is not within the range of 'type-name' (C++)

You have attempted to assign a value that is out of the range of the specified type.

3.1.2.283 E2186: Unexpected end of file in comment started on line 'number' (C++)

The source file ended in the middle of a comment.

This is normally caused by a missing close of comment (*/).

3.1.2.284 E2187: Unexpected end of file in conditional started on line 'number' (C++)

The source file ended before the compiler (or MAKE) encountered `#endif`.

The `#endif` either was missing or misspelled.

Every `#if` statement needs a matching `#endif` statement.

3.1.2.285 E2188: Expression syntax (C++)

This is a catch-all error message when the compiler parses an expression and encounters a serious error.

Possible Causes

This is most commonly caused by one of the following:

- two consecutive operators
- mismatched or missing parentheses
- a missing semicolon on the previous statement.

Solutions

If the line where the error occurred looks syntactically correct, look at the line directly above for errors.

Try moving the line with the error to a different location in the file and recompiling.

If the error still occurs at the moved statement, the syntax error is occurring somewhere in that statement.

If the error occurred in another statement, the syntax error is probably in the surrounding code.

3.1.2.286 E2190: Unexpected closing brace (C++)

An extra right brace was encountered where none was expected. Check for a missing closing brace.

Useful Tip:

The IDE has a mechanism for finding a matching curly brace. If you put the cursor on the '{' or '}' character, hold down Ctrl, hit 'Q' and then '{' or '}', it will position the cursor on the matching brace.

3.1.2.287 E2189: extern variable cannot be initialized (C++)

The storage class extern applied to a variable means that the variable is being declared but not defined here--no storage is being allocated for it.

Therefore, you can't initialize the variable as part of the declaration.

3.1.2.288 E2344: Earlier declaration of 'identifier' (C++)

This error message only shows up after the messages "Multiple declaration for 'identifier'" and "Type mismatch in redeclaration of 'identifier'". It tells you where the previous definition of the identifier in question was found by the compiler, so you don't have to search for it.

3.1.2.289 E2192: Too few parameters in call (C++)

This error message occurs when a call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given. Make certain that your call to a function has the same parameters as the function prototype.

3.1.2.290 E2193: Too few parameters in call to 'function' (C++)

A call to the named function (declared using a prototype) has too few arguments.

Make certain that the parameters in the call to the function match the parameters of the function prototype.

3.1.2.291 E2194: Could not find file 'filename' (C++)

The compiler is unable to find the file supplied on the command line.

3.1.2.292 E2197: File name too long (C++)

The file name given in an `#include` directive was too long for the compiler to process.

File names in DOS must be no more than 79 characters long.

3.1.2.293 E2195: Cannot evaluate function call (C++)

The error message is issued if someone tries to explicitly construct an object or call a virtual function.

In integrated debugger expression evaluation, calls to certain functions (including implicit conversion functions, constructors, destructors, overloaded operators, and inline functions) are not supported.

3.1.2.294 E2198: Not a valid expression format type (C++)

Invalid format specifier following expression in the debug evaluate or watch window. A valid format specifier is an optional repeat value followed by a format character (c, d, f[n], h, x, m, p, r, or s).

3.1.2.295 E2200: Functions may not be part of a struct or union (C++)

This C struct or union field was declared to be of type function rather than pointer to function.

Functions as fields are allowed only in C++.

3.1.2.296 Floating point error: Divide by 0 OR Floating point error: Domain OR Floating point error: Overflow (C++)

These fatal errors result from a floating-point operation for which the result is not finite:

- Divide by 0 means the result is +INF or -INF exactly, such as 1.0/0.0.
- Domain means the result is NAN (not a number), like 0.0/0.0.
- Overflow means the result is +INF (infinity) or -INF with complete loss of precision, such as assigning 1e200*1e200 to a double.

3.1.2.297 Floating point error: Stack fault (C++)

The floating-point stack has been overrun. This error may be due to assembly code using too many registers or due to a misdeclaration of a floating-point function.

The program prints the error message and calls abort and _exit.

These floating-point errors can be avoided by masking the exception so that it doesn't occur, or by catching the exception with signal.

3.1.2.298 Floating point error: Partial loss of precision OR Floating point error: Underflow (C++)

These exceptions are masked by default, because underflows are converted to zero and losses of precision are ignored.

3.1.2.299 E2201: Too much global data defined in file (C++)

The sum of the global data declarations exceeds 64K bytes. This includes any data stored in the DGROUP (all global variables, literal strings, and static locals).

Solutions

Check the declarations for any array that might be too large. You can also remove variables from the DGROUP.

Here's how:

- Declare the variables as automatic. This uses stack space.
- Dynamically allocate memory from the heap using calloc, malloc, or farmalloc for the variables. This requires the use of pointers.

Literal strings are also put in the DGROUP. Get the file farstr.zip from our BBS to extract literal strings into their own segment.

3.1.2.300 E2203: Goto bypasses initialization of a local variable (C++)

In C++, it is illegal to bypass the initialization of a local variable.

This error indicates a goto statement that can transfer control past this local variable.

3.1.2.301 E2204: Group overflowed maximum size: 'name' (C++)

The total size of the segments in a group (for example, DGROUP) exceeded 64K.

3.1.2.302 E2206: Illegal character 'character' (0x'value') (C++)

The compiler encountered some invalid character in the input file.

The hexadecimal value of the offending character is printed.

This can also be caused by extra parameters passed to a function macro.

3.1.2.303 E2207: Implicit conversion of 'type1' to 'type2' not allowed (C++)

When a member function of a class is called using a pointer to a derived class, the pointer value must be implicitly converted to point to the appropriate base class.

In this case, such an implicit conversion is illegal.

3.1.2.304 E2208: Cannot access an inactive scope (C++)

You have tried to evaluate or inspect a variable local to a function that is currently not active. (This is an integrated debugger expression evaluation message.)

3.1.2.305 E2209: Unable to open include file 'filename' (C++)

The compiler could not find the named file.

Possible Causes

- The named file does not exist.
- An `#include` file included itself.
- You do not have FILES set in CONFIG.SYS on your root directory.

Solutions

- Verify that the named file exists.
- Set FILES = 20 in CONFIG.SYS.

3.1.2.306 E2210: Reference member 'member' is not initialized (C++)

References must always be initialized, in the constructor for the class.

A class member of reference type must have an initializer provided in all constructors for that class.

This means you can't depend on the compiler to generate constructors for such a class, because it has no way of knowing how to initialize the references.

3.1.2.307 E2212: Function defined inline after use as extern (C++)

Functions can't become inline after they have already been used.

Either move the inline definition forward in the file or delete it entirely.

The compiler encountered something like:

```
myex();
twoex() { myex(); }
inline myex() { return 2; } // error
```

and already used the function as an extern before it saw that it was specified as inline. This would be correct:

```
myex();
inline myex() { return 2; }
twoex() { myex(); }
```

or better:

```
inline myex();
inline myex() { return 2; }
twoex() { myex(); }
```

3.1.2.308 E2211: Inline assembly not allowed in inline and template functions (C++)

The compiler can't handle inline assembly statements in a C++ inline or template function.

You could eliminate the inline assembly code or, in the case of an inline function, make this a macro, and remove the inline storage class.

3.1.2.309 F1001: Internal code generator error (C++)

An error has occurred in the internal logic of the code generator. Contact CodeGear technical support.

3.1.2.310 E2413: Invalid template declaration (C++)

After the declarator of a template member, either a semicolon, an initialization, or a body was expected, but some other, illegal token was found. This message appears when a template member is declared outside of the template, but the syntax was wrong.

3.1.2.311 E2070: Invalid use of namespace 'identifier' (C++)

A namespace identifier was used in an illegal way, for example, in an expression.

3.1.2.312 E2214: Cannot have a 'non-inline function/static data' in a local class (C++)

All members of classes declared local to a function must be entirely defined in the class definition.

This means that local classes cannot contain any static data members, and all of their member functions must have bodies defined within the class definition.

3.1.2.313 E2215: Linkage specification not allowed (C++)

Linkage specifications such as extern "C" are only allowed at the file level.

Move this function declaration out to the file level.

3.1.2.314 E2216: Unable to create turboc.$ln (C++)

The compiler cannot create the temporary file TURBOC.$LN because it cannot access the disk or the disk is full.

3.1.2.315 E2218: Templates can only be declared at namespace or class scope (C++)

Templates cannot be declared inside classes or functions. They are only allowed in the global scope, or file level.

For example:

```
void func()
{
  template <class T> myClass {  // Error
    T i;
  };
}
```

3.1.2.316 E2217: Local data exceeds segment size limit (C++)

The local variables in the current function take up more than 64K.

3.1.2.317 E2219: Wrong number of arguments in call of macro 'macro' (C++)

Your source file called the named macro with an incorrect number of arguments.

3.1.2.318 E2220: Invalid macro argument separator (C++)

In a macro definition, arguments must be separated by commas.

The compiler encountered some other character after an argument name.

This is correct:

```
#define tri_add(a, b, c)  ((a) + (b) + (c))
```

This is incorrect:

```
#define tri_add(a  b. c)  ((a) + (b) + (c))
```

3.1.2.319 E2221: Macro argument syntax error (C++)

An argument in a macro definition must be an identifier.

The compiler encountered some non-identifier character where an argument was expected.

3.1.2.320 E2222: Macro expansion too long (C++)

A macro can't expand to more than 4,096 characters.

3.1.2.321 E2223: Too many decimal points (C++)

The compiler encountered a floating-point constant with more than one decimal point.

3.1.2.322 E2224: Too many exponents (C++)

The compiler encountered more than one exponent in a floating-point constant.

3.1.2.323 E2225: Too many initializers (C++)

The compiler encountered more initializers than were allowed by the declaration being initialized.

3.1.2.324 E2226: Extra parameter in call (C++)

A call to a function, via a pointer defined with a prototype, had too many arguments.

3.1.2.325 E2227: Extra parameter in call to function (C++)

A call to the named function (which was defined with a prototype) had too many arguments given in the call.

3.1.2.326 E2228: Too many error or warning messages (C++)

There were more errors or warnings than allowed.

3.1.2.327 E2233: Cannot initialize a class member here (C++)

Individual members of structs, unions, and C++ classes can't have initializers.

A struct or union can be initialized as a whole using initializers inside braces.

A C++ class can only be initialized by the use of a constructor.

3.1.2.328 E2232: Constant/Reference member 'member' in class without constructors (C++)

A class that contains constant or reference members (or both) must have at least one user-defined constructor.

Otherwise, there would be no way to ever initialize such members.

3.1.2.329 E2229: Member 'member' has the same name as its class (C++)

A static data member, enumerator, member of an anonymous union, or nested type cannot have the same name as its class.

Only a member function or a non-static member can have a name that is identical to its class.

3.1.2.330 E2234: Memory reference expected (C++)

The built-in assembler requires a memory reference.

You probably forgot to put square brackets around an index register operand.

3.1.2.331 E2231: Member 'member' cannot be used without an object (C++)

This means that you have written class::member where 'member' is an ordinary (non-static) member, and there is no class to associate with that member.

For example, it is legal to write this:

```
obj.class::member
```

but not to write this:

```
class::member
```

3.1.2.332 E2235: Member function must be called or its address taken (C++)

A reference to a member function must be called, or its address must be taken with & operator.

In this case, a member function has been used in an illegal context.

For example:

```
class A
{
   typedef int (A::* infptr)(void);
public:
   A();
   int myex(void);
   int three;
} a;
A::A()
{
   infptr one = myex;       //illegal - call myex or take address?
   infptr two = &A::myex;   //correct
   three = (a.*one)() + (a.*two)();
}
```

3.1.2.333 O2237: DPMI programs must use the large memory model (C++)

The compiler no longer issues this error.

3.1.2.334 E2238: Multiple declaration for 'identifier' (C++)

This identifier was improperly declared more than once.

This might be caused by conflicting declarations such as:

- int a; double a;
- a function declared two different ways, or
- a label repeated in the same function, or
- some declaration repeated other than an extern function or a simple variable

This can also happen by inadvertently including the same header file twice. For example, given:

```
//a.h
struct A { int a; };
//b.h
#include "a.h"
//myprog.cpp
#include "a.h"
#include "b.h"
```

myprog.cpp will get two declarations for the struct A. To protect against this, one would write the a.h header file as:

```
//a.h
#ifndef __A_H
#define __A_H
struct A { int a; };
#endif
```

This will allow one to safely include a.h several times in the same source code file.

3.1.2.335 E2239: 'identifier' must be a member function (C++)

Most C++ operator functions can be members of classes or ordinary non-member functions, but these are required to be members of classes:

- operator =
- operator ->
- operator ( )
- type conversions

This operator function is not a member function but should be.

3.1.2.336 E2240: Conversion of near pointer not allowed (C++)

A near pointer cannot be converted to a far pointer in the expression evaluation box when a program is not currently running. This is because the conversion needs the current value of DS in the user program, which doesn't exist.

3.1.2.337 E2243: Array allocated using 'new' may not have an initializer (C++)

When initializing a vector (array) of classes, you must use the constructor that has no arguments.

This is called the default constructor, which means that you can't supply constructor arguments when initializing such a vector.

3.1.2.338 E2244: 'new' and 'delete' not supported (C++)

The integrated debugger does not support the evaluation of the new and delete operators.

3.1.2.339 E2245: Cannot allocate a reference (C++)

You have attempted to create a reference using the new operator.

This is illegal, because references are not objects and can't be created through new.

3.1.2.340 E2309: Inline assembly not allowed (C++)

Your source file contains inline assembly language statements and you are compiling it from within the integrated environment.

You must use the BCC command to compile this source file from the DOS command line.

3.1.2.341 E2250: No base class to initialize (C++)

This C++ class constructor is trying to implicitly call a base class constructor, but this class was declared with no base classes.

Check your declarations.

3.1.2.342 E2254: : expected after private/protected/private (C++)

When used to begin a private, protected, or public section of a C++ class, the reserved words "private," "protected," and "public" must be followed by a colon.

3.1.2.343 E2255: Use :: to take the address of a member function (C++)

If f is a member function of class c, you take its address with the syntax

```
&c::f
```

Note the use of the class type name (not the name of an object) and the :: separating the class name from the function name.

(Member function pointers are not true pointer types, and do not refer to any particular instance of a class.)

3.1.2.344 E2256: No : following the ? (C++)

The question mark (?) and colon (:) operators do not match in this expression.

The colon might have been omitted, or parentheses might be improperly nested or missing.

3.1.2.345 E2257: , expected (C++)

A comma was expected in a list of declarations, initializations, or parameters.

This problem is often caused by a missing syntax element earlier in the file or one of its included headers.

3.1.2.346 E2258: Declaration was expected (C++)

A declaration was expected here but not found.

This is usually caused by a missing delimiter such as a comma, semicolon, right parenthesis, or right brace.

3.1.2.347 E2259: Default value missing (C++)

When a C++ function declares a parameter with a default value, all of the following parameters must also have default values.

In this declaration, a parameter with a default value was followed by a parameter without a default value.

3.1.2.348 E2260: Default value missing following parameter 'parameter' (C++)

All parameters following the first parameter with a default value must also have defaults specified.

3.1.2.349 E2263: Exception handling not enabled (C++)

A 'try' block was found with the exception handling disabled.

3.1.2.350 E2264: Expression expected (C++)

An expression was expected here, but the current symbol can't begin an expression.

This message might occur where the controlling expression of an if or while clause is expected or where a variable is being initialized.

This message is often due to a symbol that is missing or has been added.

3.1.2.351 E2266: No file names given (C++)

The command line contained no file names. You must specify a source file name.

3.1.2.352 E2265: No file name ending (C++)

The file name in an `#include` statement was missing the correct closing quote or angle bracket.

3.1.2.353 E2271: Goto statement missing label (C++)

The `goto` keyword must be followed by an identifier.

3.1.2.354 E2272: Identifier expected (C++)

An identifier was expected here, but not found.

In C, an identifier is expected in the following situations:

- in a list of parameters in an old-style function header
- after the reserved words struct or union when the braces are not present, and

- as the name of a member in a structure or union (except for bit fields of width 0).

In C++, an identifier is also expected in these situations:

- in a list of base classes from which another class is derived, following a double colon (::), and
- after the reserved word "operator" when no operator symbol is present.

3.1.2.355 E2275: Opening brace expected (C++)

A left brace was expected at the start of a block or initialization.

3.1.2.356 E2276: (expected (C++)

A left parenthesis was expected before a parameter list.

3.1.2.357 E2274: < expected (C++)

The keyword `template` was not followed by <.

Every template declaration must include the template formal parameters enclosed within < >, immediately following the template keyword.

3.1.2.358 E2277: Lvalue required (C++)

The left side of an assignment operator must be an addressable expression.

Addressable expressions include the following:

- numeric or pointer variables
- structure field references or indirection through a pointer
- a subscripted array element

3.1.2.359 E2278: Multiple base classes not supported for Delphi classes (C++)

Delphi style classes cannot have multiple base classes.

Example

```
struct__declspec(delphiclass) base1  {};
struct__declspec(delphiclass) base2  {};
structderived : base1, base2         {}; // Error
```

3.1.2.360 E2280: Member identifier expected (C++)

The name of a structure or C++ class member was expected here, but not found. The right side of a dot (.) or arrow (->) operator must be the name of a member in the structure or class on the left of the operator.

3.1.2.361 E2279: Cannot find default constructor to initialize member 'identifier' (C++)

When the following occurs

1.A C++ class 'class1' contains a member of class 'class2,'

and

2.You want to construct an object of type 'class1' (but not from another object of type 'class1'). There must be a constructor class2::class2() so that the member can be constructed.

This constructor without parameters is called the default constructor.

The compiler will supply a default constructor automatically unless you have defined any constructor for class 'class2'.

In that case, the compiler will not supply the default constructor automatically (you must supply one.

3.1.2.362 E2310: Only member functions may be 'const' or 'volatile' (C++)

Something other than a class member function has been declared `const` or `volatile`.

3.1.2.363 E2311: Non-virtual function 'function' declared pure (C++)

Only virtual functions can be declared pure, because derived classes must be able to override them.

3.1.2.364 E2283: Use . or -> to call 'function' (C++)

You attempted to call a member function without providing an object. This is required to call a member function.

```
class X {
   member func() {}
};
X x;
X*xp = new X;
X.memberfunc();
Xp-> memberfunc();
```

3.1.2.365 E2284: Use . or -> to call 'member', or & to take its address (C++)

A reference to a non-static class member without an object was encountered.

Such a member can't be used without an object, or its address must be taken with the & operator.

3.1.2.366 E2285: Could not find a match for 'argument(s)' (C++)

No C++ function could be found with parameters matching the supplied arguments. Check parameters passed to function or

overload function for parameters that are being passed.

3.1.2.367 E2286: Overloaded function resolution not supported (C++)

In integrated debugger expression evaluation, resolution of overloaded functions or operators is not supported, not even to take an address.

3.1.2.368 E2287: Parameter 'number' missing name (C++)

In a function definition header, this parameter consisted only of a type specifier 'number' with no parameter name.

This is not legal in C.

(It is allowed in C++, but there's no way to refer to the parameter in the function.)

3.1.2.369 E2288: Pointer to structure required on left side of -> or ->* (C++)

Nothing but a pointer is allowed on the left side of the arrow (->) in C or C++.

In C++ a -> operator is allowed.

3.1.2.370 E2290: 'code' missing] (C++)

This error is generated if any of the following occur:

- Your source file declared an array in which the array bounds were not terminated by a right bracket.
- The array specifier in an operator is missing a right bracket.
- The operator [ ] was declared as operator [.
- A right bracket is missing from a subscripting expression.

Add the bracket or fix the declaration.

Check for a missing or extra operator or mismatched parentheses.

3.1.2.371 E2291: brace expected (C++)

A right brace was expected at the end of a block or initialization.

3.1.2.372 E2292: Function should return a value (C++)

Your source file declared the current function to return some type other than `int` or `void`, but the compiler encountered a return with no value. This usually indicates some sort of error.

Functions declared as returning `int` are exempt because older versions of C did not support `void` function return types.

3.1.2.373 E2293:) expected (C++)

A right parenthesis was expected at the end of a parameter list.

3.1.2.374 E2294: Structure required on left side of . or .* (C++)

The left side of a dot (.) operator (or C++ dot-star operator, .*) must evaluate to a structure type. In this case it did not.

This error can occur when you create an instance of a class using empty parentheses, and then try to access a member of that 'object'.

3.1.2.375 E2312: 'constructor' is not an unambiguous base class of 'class' (C++)

A C++ class constructor is trying to call a base class constructor 'constructor.'

This error can also occur if you try to change the access rights of 'class::constructor.'

Check your declarations.

3.1.2.376 E2313: Constant expression required (C++)

Arrays must be declared with constant size.

This error is commonly caused by misspelling a `#define` constant.

3.1.2.377 E2296: Templates not supported (C++)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

3.1.2.378 E2314: Call of nonfunction (C++)

The name being called is not declared as a function.

This is commonly caused by incorrectly declaring the function or misspelling the function name.

3.1.2.379 E2321: Declaration does not specify a tag or an identifier (C++)

This declaration doesn't declare anything.

This may be a struct or union without a tag or a variable in the declaration. C++ requires that something be declared.

For example:

```
struct
{
int a
};
//no tag or identifier
```

3.1.2.380 E2297: 'this' can only be used within a member function (C++)

In C++, "this" is a reserved word that can be used only within class member functions.

3.1.2.381 E2316: 'identifier' is not a member of 'struct' (C++)

You are trying to reference 'identifier' as a member of 'struct', but it is not a member.

Check your declarations.

3.1.2.382 E2317: 'identifier' is not a parameter (C++)

In the parameter declaration section of an old-style function definition, 'identifier' is declared but not listed as a parameter. Either remove the declaration or add 'identifier' as a parameter.

3.1.2.383 E2319: 'identifier' is not a public base class of 'classtype' (C++)

The right operand of a .*, ->*, or ::operator was not a pointer to a member of a class that is either identical to (or an unambiguous accessible base class of) the left operand's class type.

3.1.2.384 E2320: Expression of scalar type expected (C++)

The !, ++, and -- operators require an expression of scalar type.

Only these types are allowed:

- char
- short
- int
- long
- enum
- float
- double
- long double
- pointer

3.1.2.385 E2302: No type information (C++)

The integrated debugger has no type information for this variable. Ensure that you've compiled the module with debug information. If it has, the module may have been compiled by another compiler or assembler.

3.1.2.386 E2303: Type name expected (C++)

One of these errors has occurred:

- In declaring a file-level variable or a struct field, neither a type name nor a storage class was given.
- In declaring a typedef, no type for the name was supplied.
- In declaring a destructor for a C++ class, the destructor name was not a type name (it must be the same name as its class).
- In supplying a C++ base class name, the name was not the name of a class.

3.1.2.387 E2304: 'Constant/Reference' variable 'variable' must be initialized (C++)

This C++ object is declared constant or as a reference, but is not initialized.

It must be initialized at the point of declaration.

3.1.2.388 E2305: Cannot find 'class::class' ('class'&) to copy a vector OR Cannot find 'class'::operator=('class'&) to copy a vector (C++)

When a C++ class 'class1' contains a vector (array) of class 'class2', and you want to construct an object of type 'class1' from another object of type 'class 1', you must use this constructor:

```
class2::class2(class2&)
```

so that the elements of the vector can be constructed.

The constructor, called a copy constructor, takes just one parameter (which is a reference to its class).

Usually, the compiler supplies a copy constructor automatically.

However, if you have defined a constructor for class 'class2' that has a parameter of type 'class2&' and has additional parameters with default values, the copy constructor can't exist and can't be created by the compiler.

This is because these two can't be distinguished:

```
class2::class2(class2&)
class2::class2(class2&, int = 1)
```

You must redefine this constructor so that not all parameters have default values.

You can then define a reference constructor or let the compiler create one.

Cannot find class::operator= ...

When a C++ class 'class1' contains a vector (array) of class 'class2', and you want to copy a class of type 'class1', you must use this assignment operator:

```
class2::class2(class2&)
```

so that the elements of the vector can be copied.

Usually, the compiler automatically supplies this operator.

However, if you have defined an operator= for class 'class2' that does not take a parameter of type 'class2&,' the compiler will not supply it automatically--you must supply one.

3.1.2.389 E2306: Virtual base classes not supported for Delphi classes (C++)

Delphi style classes cannot be derived virtually, not even from other Delphi style classes.

Example

```
struct __declspec(delphiclass) base  {};
struct derived : virtual base        {};    // Error
```

3.1.2.390 E2308: do statement must have while (C++)

Your source file contained a do statement that was missing the closing while keyword.

3.1.2.391 E2322: Incorrect number format (C++)

The compiler encountered a decimal point in a hexadecimal number.

3.1.2.392 E2324: Numeric constant too large (C++)

String and character escape sequences larger than hexadecimal or octal 77 can't be generated.

Two-byte character constants can be specified by using a second backslash. For example,

```
\\
```

represents a two-byte constant.

A numeric literal following an escape sequence should be broken up like this:

```
printf("\x0A" "12345");
```

This prints a carriage return followed by 12345.

3.1.2.393 E2282: Namespace name expected (C++)

The name of a namespace symbol was expected.

3.1.2.394 E2334: Namespace member 'identifier' declared outside its namespace (C++)

Namespace members must be declared inside their namespace. You can only use explicit qualification to define a namespace

member (for example, to give a body for a function declared in a namespace). The declaration itself must be inside the namespace.

3.1.2.395 E2325: Illegal octal digit (C++)

The compiler found an octal constant containing a non-octal digit (8 or 9).

3.1.2.396 E2329: Invalid combination of opcode and operands (C++)

The built-in assembler does not accept this combination of operands.

Possible causes

- There are too many or too few operands for this assembler opcode.
- The number of operands is correct, but their types or order do not match the opcode.

3.1.2.397 E2327: Operators may not have default argument values (C++)

It is illegal for overloaded operators to have default argument values.

3.1.2.398 E2330: Operator must be declared as function (C++)

An overloaded operator was declared with something other than function type.

For example:

```
class A
{
   A& operator +;   ..note missing parenthesis
};
```

In the example, the function operator '()' is missing, so the operator does not have function type and generates this error.

3.1.2.399 E2333: Class member 'member' declared outside its class (C++)

C++ class member functions can be declared only inside the class declaration.

Unlike nonmember functions, they can't be declared multiple times or at other locations.

3.1.2.400 E2335: Overloaded 'function name' ambiguous in this context (C++)

The only time an overloaded function name can be used or assigned without actually calling the function is when a variable or parameter of the correct function pointer type is initialized or assigned the address of the overload function.

In this case, an overloaded function name has been used in some other context, for example, the following code will generate this error:

```
class A{
   A(){myex;}           //calling the function
   void myex(int) {}    //or taking its address?
   void myex(float){}
};
```

3.1.2.401 E2339: Cannot overload 'main' (C++)

You cannot overload main.

3.1.2.402 E2336: Pointer to overloaded function 'function' doesn't match 'type' (C++)

A variable or parameter is assigned (or initialized with) the address of an overloaded function.

However, the type of the variable or parameter doesn't match any of the overloaded functions with the specified name.

3.1.2.403 E2337: Only one of a set of overloaded functions can be "C" (C++)

C++ functions are by default overloaded, and the compiler assigns a new name to each function.

If you wish to override the compiler's assigning a new name by declaring the function `extern` "C", you can do this for only one of a set of functions with the same name.

(Otherwise the linker would find more than one global function with the same name.)

3.1.2.404 E2338: Overlays only supported in medium, large, and huge memory models (C++)

The compiler no longer issues this error.

3.1.2.405 E2340: Type mismatch in parameter 'number' (C++)

The function called, via a function pointer, was declared with a prototype.

However, the given parameter number (counting left to right from 1) could not be converted to the declared parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

3.1.2.406 E2341: Type mismatch in parameter 'number' in call to 'function' (C++)

Your source file declared the named function with a prototype, and the given parameter number (counting left to right from 1) could not be converted to the declared parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'", but the mismatch might be due to many other reasons.

3.1.2.407 E2342: Type mismatch in parameter 'parameter' (C++)

Your source file declared the function called via a function pointer with a prototype.

However, the named parameter could not be converted to the declared parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

3.1.2.408 E2343: Type mismatch in parameter 'parameter' in call to 'function' (C++)

Your source file declared the named function with a prototype, and the named parameter could not be converted to the declared parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

3.1.2.409 E2345: Access can only be changed to public or protected (C++)

A C++ derived class can modify the access rights of a base class member, but only to public or protected.

A base class member can't be made private.

3.1.2.410 E2349: Nonportable pointer conversion (C++)

An implicit conversion between a pointer and an integral type is required, but the types are not the same size. You must use an explicit cast.

This conversion might not make any sense, so be sure this is what you want to do.

3.1.2.411 E2350: Cannot define a pointer or reference to a reference (C++)

It is illegal to have a pointer to a reference or a reference to a reference.

3.1.2.412 E2352: Cannot create instance of abstract class 'class' (C++)

Abstract classes (those with pure virtual functions) can't be used directly, only derived from.

When you derive an abstract base class, with the intention to instantiate instances of this derived class, you must override each of the pure virtual functions of the base class exactly as they are declared.

For example:

```
class A {
public:
   virtual myex( int ) = 0;
   virtual twoex( const int ) const = 0;
};
class B : public A {
public:
   myex( int );
   twoex( const int );
};
B b;   // error
```

The error occurs because we have not overridden the virtual function in which twoex can act on const objects of the class. We have created a new one which acts on non-const objects. This would compile:

```
class A {
public:
   virtual myex( int ) = 0;
   virtual twoex( const int ) const = 0;
};
class B : public A {
public:
   myex( int );
   twoex( const int ) const;
};
B b;   // ok
```

3.1.2.413 E2354: Two operands must evaluate to the same type (C++)

The types of the expressions on both sides of the colon in the conditional expression operator (?:) must be the same, except for the usual conversions.

These are some examples of usual conversions

- `char` to `int`
- `float` to `double`
- void* to a particular pointer

In this expression, the two sides evaluate to different types that are not automatically converted.

This might be an error or you might merely need to cast one side to the type of the other.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

3.1.2.414 E2355: Recursive template function: 'x' instantiated 'y' (C++)

The compiler has detected a recursive template function instance. For example:

```
template<class T> void f(T x)
{
   f((T*)0);   // recursive template function!
}
void main()
{
   f(0);
}
```

The compiler issue one message for each nesting of the recursive instantiation, so it is usually obvious where the recursion has occurred. To fix a recursive template, either change the dependencies or provide a specialized version that will stop the recursion. For example, adding the following function definition to the above program will remove the endless recursion:

```
void f(int **)
{
}
```

3.1.2.415 E2356: Type mismatch in redeclaration of 'identifier' (C++)

Your source file redeclared a variable with a different type than was originally declared for the variable.

Possible Causes

This can occur if a function is called and subsequently declared to return something other than an integer.

Solutions

If this has happened, you must declare the function before the first call to it.

3.1.2.416 E2357: Reference initialized with 'type1', needs lvalue of type 'type2' (C++)

A reference variable that is not declared constant must be initialized with an lvalue of the appropriate type.

In this case, the initializer either wasn't an lvalue, or its type didn't match the reference being initialized.

3.1.2.417 E2358: Reference member 'member' needs a temporary for initialization (C++)

You provided an initial value for a reference type that was not an lvalue of the referenced type.

This requires the compiler to create a temporary for the initialization.

Because there is no obvious place to store this temporary, the initialization is illegal.

3.1.2.418 E2360: Invalid register combination (e.g. [BP+BX]) (C++)

The built-in assembler detected an illegal combination of registers in an instruction.

These are valid index register combinations:

- [BX]
- [BP]
- [SI]
- [DI]
- [BX+SI]
- [BX+DI]
- [BP+SI]
- [BP+DI]

Other index register combinations are not allowed.

3.1.2.419 E2361: 'specifier' has already been included (C++)

This type specifier occurs more than once in this declaration.

Delete or change one of the occurrences.

3.1.2.420 E2362: Repeat count needs an lvalue (C++)

The expression before the comma (,) in the Watch or Evaluate window must be an accessible region of storage. For example, expressions like this one are not valid:

```
i++,10d
x = y, 10m
```

3.1.2.421 E2363: Attempting to return a reference to local variable 'identifier' (C++)

This C++ function returns a reference type, and you are trying to return a reference to a local (auto) variable.

This is illegal, because the variable referred to disappears when the function exits.

You can return a reference to any static or global variable, or you can change the function to return a value instead.

3.1.2.422 E2364: Attempting to return a reference to a local object (C++)

You attempted to return a reference to a temporary object in a function that returns a reference type. This may be the result of a constructor or a function call.

This object will disappear when the function returns, making the reference illegal.

3.1.2.423 E2365: Member pointer required on right side of .* or ->* (C++)

The right side of a C++ dot-star (.*) or an arrow star (->*) operator must be declared as a pointer to a member of the class specified by the left side of the operator.

In this case, the right side is not a member pointer.

3.1.2.424 E2366: Can't inherit non-RTTI class from RTTI base OR E2367 Can't inherit RTTI class from non-RTTI base (C++)

When virtual functions are present, the RTTI attribute of all base classes must match that of the derived class.

3.1.2.425 E2368: RTTI not available for expression evaluation (C++)

Expressions requiring RTTI are not supported by the expression evaluator in the integrated debugger. This error message is only issued by the expression evaluator (if you try to Inspect, Watch, or Evaluate), not by the compiler.

3.1.2.426 E2371: sizeof may not be applied to a bit field (C++)

sizeof returns the size of a data object in bytes, which does not apply to a bit field.

3.1.2.427 E2372: sizeof may not be applied to a function (C++)

sizeof can be applied only to data objects, not functions.

You can request the size of a pointer to a function.

3.1.2.428 E2373: Bit field cannot be static (C++)

Only ordinary C++ class data members can be declared static, not bit fields.

3.1.2.429 E2374: Function 'function' cannot be static (C++)

Only ordinary member functions and the operators new and delete can be declared static.

Constructors, destructors and other operators must not be static.

3.1.2.430 Stack overflow (C++)

This error is reported when you compile a function with the Test Stack Overflow option on, but there is not enough stack space to

allocate the function's local variables.

This error can also be caused by the following:

- infinite recursion, or
- an assembly language procedure that does not maintain the stack project
- a large array in a function

3.1.2.431 E2376: statement missing (C++)

In a do, for, if, switch, or while statement, the compiler found no left parenthesis after the while keyword or test expression.

3.1.2.432 E2377: statement missing) (C++)

In a do, for, if, switch, or while statement, the compiler found no right parenthesis after the while keyword or test expression.

3.1.2.433 E2378: do-while or for statement missing ; (C++)

In a do or for statement, the compiler found no semicolon after the right parenthesis.

3.1.2.434 E2379: Statement missing ; (C++)

The compiler encountered an expression statement without a semicolon following it.

3.1.2.435 E2380: Unterminated string or character constant (C++)

The compiler found no terminating quote after the beginning of a string or character constant.

3.1.2.436 E2381: Structure size too large (C++)

Your source file declared a structure larger than 64K.

3.1.2.437 E2382: Side effects are not allowed (C++)

Side effects such as assignments, ++, or -- are not allowed in the debugger watch window. A common error is to use x = y (not allowed) instead of x == y to test the equality of x and y.

3.1.2.438 E2383: Switch selection expression must be of integral type (C++)

The selection expression in parentheses in a `switch` statement must evaluate to an integral type (`char, short, int, long, enum`).

You might be able to use an explicit cast to satisfy this requirement.

3.1.2.439 E2433: Specialization after first use of template (C++)

An ANSI C++ rule requires that a specialization for a function template be declared before its first use. This error message is only issued when the ANSI conformance option (-A) is active.

3.1.2.440 E2384: Cannot call near class member function with a pointer of type 'type' (C++)

Also E2385 Cannot call near class member function 'function' with a pointer of type 'type'

Member functions of near classes can't be called via a member pointer.

This also applies to calls using pointers to members.

(Remember, classes are near by default in the tiny, small, and medium memory models.)

Either change the pointer to be near, or declare the class as far.

3.1.2.441 E2390: Type mismatch in parameter 'number' in template class name 'template' (C++)

The actual template argument value supplied for the given parameter did not exactly match the formal template parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

3.1.2.442 E2391: Type mismatch in parameter 'parameter' in template class name 'template' (C++)

The actual template argument value supplied for the given parameter did not exactly match the formal template parameter type.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

3.1.2.443 E2394: Too few arguments passed to template 'template' (C++)

A template class name was missing actual values for some of its formal parameters.

3.1.2.444 E2395: Too many arguments passed to template 'template' (C++)

A template class name specified too many actual values for its formal parameters.

3.1.2.445 E2396: Template argument must be a constant expression (C++)

A non-type template class argument must be a constant expression of the appropriate type.

This includes constant integral expressions and addresses of objects or functions with external linkage or members.

3.1.2.446 E2401: Invalid template argument list (C++)

This error indicates that an illegal template argument list was found.

In a template declaration, the keyword `template` must be followed by a list of formal arguments enclosed within < and > delimiters.

3.1.2.447 E2400: Nontype template argument must be of scalar type (C++)

A nontype formal template argument must have scalar type; it can have an integral, enumeration, or pointer type.

3.1.2.448 E2415: Template functions may only have 'type-arguments' (C++)

A function template was declared with a non-type argument.

This is not allowed with a template function, as there is no way to specify the value when calling it.

3.1.2.449 E2425: 'member' is not a valid template type member (C++)

A member of a template with some actual arguments that depend on the formal arguments of an enclosing template was found not to be a member of the specified template in a particular instance.

3.1.2.450 E2428: Templates must be classes or functions (C++)

The declaration in a template declaration must specify either a class type or a function.

3.1.2.451 E2432: 'template' qualifier must name a template class or function instance' (C++)

When defining a template class member, the actual arguments in the template class name used as the left operand for the :: operator must match the formal arguments of the template class.

3.1.2.452 E2442: Two consecutive dots (C++)

Because an ellipsis contains three dots (...), and a decimal point or member selection operator uses one dot (.), two consecutive dots cannot legally occur in a C program.

3.1.2.453 E2443: Base class 'class' is initialized more than once (C++)

In a C++ class constructor, the list of initializations following the constructor header includes base class 'class' more than once.

3.1.2.454 E2444: Member 'member' is initialized more than once (C++)

In a C++ class constructor, the list of initializations following the constructor header includes the same member name more than once.

3.1.2.455 E2445: Variable 'identifier' is initialized more than once (C++)

This variable has more than one initialization. It is legal to declare a file level variable more than once, but it can have only one initialization (even if two are the same).

3.1.2.456 E2446: Function definition cannot be a typedef'ed declaration (C++)

In ANSI C, a function body cannot be defined using a typedef with a function Type.

Redefine the function body.

3.1.2.457 E2132: Templates and overloaded operators cannot have C linkage (C++)

You tried to use a linkage specification with a template or overloaded operator. The most common cause for this error message is having the declaration wrapped in an extern "C" linkage specification.

3.1.2.458 E2447: 'identifier' must be a previously defined enumeration tag (C++)

This declaration is attempting to reference 'ident' as the tag of an enum type, but it has not been so declared.

Correct the name, or rearrange the declarations.

3.1.2.459 E2448: Undefined label 'identifier' (C++)

The named label has a `goto` in the function, but no label definition.

3.1.2.460 E2449: Size of 'identifier' is unknown or zero (C++)

This identifier was used in a context where its size was needed.

A struct tag might only be declared (the struct not defined yet), or an extern array might be declared without a size.

It's illegal then to have some references to such an item (like sizeof) or to dereference a pointer to this type.

Rearrange your declaration so that the size of 'identifier' is available.

3.1.2.461 E2450: Undefined structure 'structure' (C++)

The named structure was used in the source file, probably on a pointer to a structure, but had no definition in the source file.

This is probably caused by a misspelled structure name or a missing declaration.

3.1.2.462 E2451: Undefined symbol 'identifier' (C++)

The named identifier has no declaration.

Possible Causes

- Actual declaration of identifier has been commented out.
- Misspelling, either at this point or at the declaration.
- An error in the declaration of the identifier.
- The header file in which the identifier is declared was not included using `#include`

Tools to help track down the problem:

GREP (see page 170)

3.1.2.463 E2453: Size of the type 'identifier' is unknown or zero (C++)

This type was used in a context where its size was needed.

For example, a struct tag might only be declared (the struct not defined yet).

It's illegal then to have some references to such an item (like sizeof) or to dereference a pointer to this type.

Rearrange your declarations so that the size of this type is available.

3.1.2.464 E2452: Size of the type is unknown or zero (C++)

This error message indicates that an array of unspecified dimension nested within another structure is initialized and the `-A` (ANSI) switch is on. For example:

```
struct
{
char a[]; //Size of 'a' is unknown or zero
}
b = { "hello" }; //Size of the type is
//unknown or zero
```

3.1.2.465 E2454: union cannot be a base type (C++)

A union can't be used as a base type for another class type.

3.1.2.466 E2455: union cannot have a base type (C++)

In general, a C++ class can be of union type, but such a class can't be derived from any other class.

3.1.2.467 E2456: Union member 'member' is of type class with 'constructor' (or destructor, or operator =) (C++)

A union can't contain members that are of type class with user-defined constructors, destructors, or operator =.

3.1.2.468 E2461: '%s' requires run-time initialization/finalization (C++)

This message is issued when a global variable that is declared as __thread (a Win32-only feature) or a static data member of a template class is initialized with a non-constant initial value.

This message is also issued when a global variable that is declared as __thread (a Win32-only feature) or a static data member of a template class has the type class with constructor or destructor.

3.1.2.469 E2464: 'virtual' can only be used with member functions (C++)

A data member has been declared with the `virtual` specifier.

Only member functions can be declared virtual.

For example:

```
class myclass
{
public:
```

```
    virtual int a;    //error
};
```

3.1.2.470 E2465: unions cannot have virtual member functions (C++)

A union can't have virtual functions as its members.

3.1.2.471 E2466: void & is not a valid type (C++)

A reference always refers to an object, but an object cannot have the type void.

Thus, the type void is not allowed.

3.1.2.472 E2467: 'Void function' cannot return a value (C++)

A function with a return type void contains a return statement that returns a value; for example, an int.

Default = displayed

3.1.2.473 E2468: Value of type void is not allowed (C++)

A value of type `void` is really not a value at all, so it can't appear in any context where an actual value is required.

Such contexts include the following:

- the right side of an assignment
- an argument of a function
- the controlling expression of an if, for, or while statement.

3.1.2.474 E2469: Cannot use tiny or huge memory model with Windows (C++)

The compiler no longer issues this error.

3.1.2.475 E2006: CodeGuarded programs must use the large memory model and be targeted for Windows (C++)

The compiler no longer issues this error.

3.1.2.476 E2269: The function 'function' is not available (C++)

You tried to call a function that is known to the evaluator, but which was not present in the program being debugged, for example, an inline function.

3.1.2.477 E2124: Invalid function call (C++)

A requested function call failed because the function is not available in the program, a parameter cannot be evaluated, and so on. The evaluator issues this message.

3.1.2.478 E2213: Invalid 'expression' in scope override (C++)

The evaluator issues this message when there is an error in a scope override in an expression you are watching or inspecting. You can specify a symbol table, a compilation unit, a source file name, etc. as the scope of the expression, and the message will appear whenever the compiler cannot access the symbol table, compilation unit, or whatever.

3.1.2.479 E2236: Missing 'identifier' in scope override (C++)

The syntax of a scope override is somehow incomplete. The evaluator issues this message.

3.1.2.480 Pure virtual function called (C++)

This is a runtime error. It is generated if the body of a pure virtual function was never generated and somehow the compiler tried to call it.

3.1.2.481 E2095: String literal not allowed in this context (C++)

This error message is issued by the evaluator when a string literal appears in a context other than a function call.

3.1.2.482 Unexpected termination during compilation [Module Seg#:offset] OR Unexpected termination during linking [Module Seg#:offset] (C++)

If either of these errors occur, it indicates a catastrophic failure of the CodeGear tools. You should contact CodeGear to report the problem and to find a potential work around for your specific case. By isolating the test case as well as possible, you will increase the chance for either CodeGear or yourself to find a work around for the problem.

Commonly, compiler failures can be worked around by moving the source code that is currently being compiled. Simple cases might be switching the order of variable declarations, or functions within the source module. Moving the scope and storage of variables also helps in many cases.

For linker failures, you can reduce the amount of debugging information that the linker has to work with. Try compiling only one or two modules with debug information instead of an entire project.

Similarly, switching the order in which object modules are handed to the linker can work around the problem. The IDE hands objects to the linker in the order that they are listed in the project tree. Try moving a source up or down in the list.

3.1.2.483 E2012: Cannot take address of 'main' (C++)

In C++, it is illegal to take the address of the main function.

3.1.2.484 E2016: Ambiguous override of virtual base member 'base_function': 'derived_function' (C++)

A virtual function in a virtual base class was overridden with two or more different functions along different paths in the inheritance hierarchy. For example,

```
struct VB
{
virtual f();
};
struct A:virtual VB
{
virtual f();
};
//{
struct B:virtual VB
virtual f();
}
```

3.1.2.485 E2021: Array must have at least one element (C++)

ANSI C and C++ require that an array be defined to have at least one element (objects of zero size are not allowed).

An old programming trick declares an array element of a structure to have zero size, then allocates the space actually needed with malloc.

You can still use this trick, but you must declare the array element to have (at least) one element if you are compiling in strict ANSI mode.

Declarations (as opposed to definitions) of arrays of unknown size are still allowed.

Example

```
char ray[];         /* definition of unknown size -- ILLEGAL */
char ray[0];        /* definition of 0 size -- ILLEGAL */
extern char ray[];  /* declaration of unknown size -- OK */
```

3.1.2.486 E2023: Array of references is not allowed (C++)

It is illegal to have an array of references, because pointers to references are not allowed and array names are coerced into pointers.

3.1.2.487 E2032: Illegal use of closure pointer (C++)

A closure pointer variable is used incorrectly. Closure variables have limited usage. For instance, you can assign a function to a closure variable, and execute that function through the closure variable, but you cannot use a closure variable like a pointer variable.

3.1.2.488 E2040: Declaration terminated incorrectly (C++)

A declaration has an extra or incorrect termination symbol, such as a semicolon placed after a function body.

A C++ member function declared in a class with a semicolon between the header and the opening left brace also generates this error.

3.1.2.489 E2047: Bad 'directive' directive syntax (C++)

A macro definition starts or ends with the ## operator, or contains the # operator that is not followed by a macro argument name.

An example of this might be:

```
Bad ifdef directive syntax
```

Note that an `#ifdef` directive must contain a single identifier (and nothing else) as the body of the directive.

Another example is:

```
Bad undef directive syntax
```

An `#undef` directive must also contain only one identifier as the body of the directive.

3.1.2.490 E2049: Class type 'type' cannot be marked as __declspec(delphireturn) (C++)

Classes marked as delphireturn are special classes that the compiler needs to recognize by name. These classes are predefined in the headers.

Some of the delphireturn classes are Variant, AnsiString, and Currency.

You cannot mark user-defined classes as delphireturn.

3.1.2.491 E2052: Dynamic function 'function' conflicts with base class 'class' (C++)

Some of the modifiers of this dynamic function conflict with the definition of the same function in the base class. The two functions should have the same modifiers. The following modifiers (among others) can cause conflicts:

- __export
- __import
- declspec(naked)
- declspec(package)
- __fastcall

3.1.2.492 E2057: Exception specification not allowed here (C++)

Function pointer type declarations are not allowed to contain exception specifications.

3.1.2.493 E2058: Exception handling variable may not be used here (C++)

An attempt has been made to use one of the exception handling values that are restricted to particular exception handling constructs, such as GetExceptionCode().

3.1.2.494 E2065: Using namespace symbol 'symbol' conflicts with intrinsic of the same name (C++)

If you define a function in a namespace, which has a name that might be replaced by a call to an intrinsic when -Oi is on, it is not permitted to have a "using" declaration which refers to that member.

For example, calls to "strcmp" are replaced by the intrinsic "__strcmp__" when -Oi is on. This means that the declaration "using N::strcmp;" would become "using N::__strcmp__", since the token replacement happens before the compiler's parser ever sees the tokens.

An error displays in this case, because the compiler doesn't know how to process "N::__strcmp__".

3.1.2.495 E2067: 'main' must have a return type of int (C++)

In C++, function main has special requirements, one of which is that it cannot be declared with any return type other than int.

3.1.2.496 E2073: Nothing allowed after pragma option pop (C++)

The #pragma option pop can only be followed by comments, blanks, or end of line.

3.1.2.497 E2091: Functions cannot return arrays or functions (C++)

A function was defined to return an array or a function. Check to see if either the intended return was a pointer to an array or function (and perhaps the * is missing) or if the function definition contained a request for an incorrect datatype.

3.1.2.498 E2093: Operator 'operator' not implemented in type 'type' for arguments of the same type (C++)

The operator you are calling is not defined in this class. When you have an expression: x + x, where x is of type class X, the operator + has to be defined in class X and be accessible.

3.1.2.499 E2094: Operator 'operator' not implemented in type 'type' for arguments of type 'type' (C++)

The operator you are calling is not defined in this class. When you have an expression: x + x, where x is of type class X, the operator + has to be defined in class X and be accessible.

3.1.2.500 E2097: Explicit instantiation only allowed at file or namespace scope (C++)

The explicit instantiation operator "template" can only be used within global or namespace scope. It cannot be used to qualify a local class or a class member, for example.

3.1.2.501 E2098: Explicit specialization declarator "template<>" now required (C++)

When specializing a function, such as providing the definition for "foo<int>", so that foo behaves specially which called for the "int" argument, now requires that the declaration begin with an explicit specialization operator.

3.1.2.502 E2099: Explicit specialization only allowed at file or namespace scope (C++)

The explicit specialization operator template<> can only be used within global or namespace scope. It cannot be used to qualify a local class or a class member, for example.

3.1.2.503 E2101: 'export' keyword must precede a template declaration (C++)

The 'export' keyword can only occur before the keyword "template" ina template declaration. It cannot be used anywhere else.

3.1.2.504 E2103: Explicit instantiation must be used with a template class or function (C++)

The explicit instantiation operator "template" can only be used to refer to templates. It cannot be used with non-templates.

3.1.2.505 E2106: Explicit specialization must be used with a template class or function (C++)

The explicit specialization operator template<> can only be used in front of a template class or function. Using it with a normal class means nothing, and hence generates an error.

3.1.2.506 E2112: Unknown unit directive: 'directive' (C++)

You cannot use this name as a unit directive. Instead use one of the following unit directives: weak, smart_init, or deny.

3.1.2.507 E2118: Bit fields must have integral type (C++)

In C++, bit fields must have an integral type. This includes enumerations.

3.1.2.508 E2120: Cannot call 'main' from within the program (C++)

C++ does not allow recursive calls of main( ).

3.1.2.509 E2125: Compiler could not generate copy constructor for class 'class' OR Compiler could not generate default constructor for class 'class' OR Compiler could not generate operator = for class 'class' (C++)

Sometimes the compiler is required to generate a member function for the user.

Whenever such a member function can't be generated due to applicable language rules, the compiler issues one of these error messages.

3.1.2.510 E2130: Circular property definition (C++)

Indicates that a property definition relies directly or indirectly on itself.

Example

```
struct pbase
{
     int __property ip1 = {read = ip2, write = ip2};
     int __property ip2 = {read = ip1, write = ip1};
};
```

The above code sample will cause this error message on any usage of ip1 or ip2.

3.1.2.511 E2131: Objects of type 'type' cannot be initialized with { } (C++)

Ordinary C structures can be initialized with a set of values inside braces.

C++ classes can only be initialized with constructors if the class has constructors, private members, functions, or base classes that are virtual.

3.1.2.512 E2148: Default argument value redeclared for parameter 'parameter' (C++)

When a parameter of a C++ function is declared to have a default value, this value can't be changed, redeclared, or omitted in any other declaration for the same function.

3.1.2.513 E2149: Default argument value redeclared (C++)

When a parameter of a C++ function is declared to have a default value, this value can't be changed, redeclared, or omitted in any other declaration for the same function.

3.1.2.514 E2151: Type mismatch in default value for parameter 'parameter' (C++)

The default parameter value given could not be converted to the type of the parameter.

The message "Type mismatch in default argument value" is used when the parameter was not given a name.

When compiling C++ programs, this message is always preceded by another message that explains the exact reason for the type mismatch.

That other message is usually "Cannot convert 'type1' to 'type2'" but the mismatch might be due to many other reasons.

3.1.2.515 E2157: Deleting an object requires exactly one conversion to pointer operator (C++)

If a person uses the 'delete' operator on an object (note: not a pointer to an object, but an object itself), the standard requires that object to define exactly one "conversion to pointer operator" which will yield the pointer that gets freed. For example:

```
char *a = new char[10];
class foo {
public:
    operator char *() { return a; }
};

int main() {
    delete[] x;
}
```

Since 'x' is not a pointer, but an object, the compiler will delete 'a', because that is what the pointer conversion operator for the object yields. Having more than one conversion to pointer operator is illegal, because the compiler would not know which one to call.

3.1.2.516 E2173: Duplicate handler for 'type1', already had 'type2' (C++)

It is not legal to specify two handlers for the same type.

3.1.2.517 E2174: The name handler must be last (C++)

In a list of catch handlers, if the specified handler is present, it must be the last handler in the list (that is, it cannot be followed by any more catch handlers).

3.1.2.518 E2177: Redeclaration of #pragma package with different arguments (C++)

You can have multiple #pragma package statements in a source file as long as they have the same arguments. This error occurs if the compiler encounters more than one #pragma package with different arguments in each.

3.1.2.519 E2178: VIRDEF name conflict for 'function' (C++)

The compiler must truncate mangled names to a certain length because of a name length limit that is imposed by the linker. This truncation may (in very rare cases) cause two names to mangle to the same linker name. If these names happen to both be VIRDEF names, the compiler issues this error message. The simplest workaround for this problem is to change the name of 'function' so that the conflict is avoided.

3.1.2.520 E2180: Dispid number already used by identifier (C++)

Dispids must be unique and the compiler checks for this.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int __fastcall foo1(void) __dispid(42);// OK
int __fastcall foo2(void) __dispid(42);// Error
};
```

3.1.2.521 E2181: Cannot override a 'dynamic/virtual' with a 'dynamic/virtual' function (C++)

When you declare a function dynamic, you cannot override this function in a derived class with a virtual function of the same name and type. Similarly when the function is declared virtual, you cannot override it with a dynamic one in a derived class.

3.1.2.522 E2202: Goto into an exception handler is not allowed (C++)

It is not legal to jump into a try block, or an exception handler that is attached to a try block.

3.1.2.523 E2205: Illegal type type in __automated section (C++)

Only certain types are allowed in __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
__automated:
int__fastcall fooInt(int);// OK
long__fastcall fooLong(long);// Error: long illegal
};
```

3.1.2.524 E2242: Specifier requires Delphi style class type (C++)

The stored, default, and nodefault storage specifiers are only allowed within property declarations of Delphi style class types.

Example

```
struct regclass
{
     int __property ip1 = { stored  = false }; // Error
     int __property ip2 = { default = 42    }; // Error
     int __property ip3 = { nodefault       }; // Error
};
struct __declspec(delphiclass) clxclass
{
     int __property ip1 = { stored  = false }; // OK
     int __property ip2 = { default = 42    }; // OK
     int __property ip3 = { nodefault       }; // OK
};
```

3.1.2.525 E2247: 'member' is not accessible (C++)

You are trying to reference C++ class member 'member,' but it is private or protected and can't be referenced from this function.

This sometimes happens when you attempt to call one accessible overloaded member function (or constructor), but the arguments match an inaccessible function.

The check for overload resolution is always made before checking for accessibility.

If this is the problem, try an explicit cast of one or more parameters to select the desired accessible function.

Virtual base class constructors must be accessible within the scope of the most derived class. This is because C++ always constructs virtual base classes first, no matter how far down the hierarchy they are. For example:

```
class A {
public:
   A();
};
class B : private virtual A {};
class C : private B {
public:
C();
};
C::C() {} // error, A::A() is not accessible
```

Since A is private to B, which is private to C, it makes A's constructor not accessible to C. However, the constructor for C must be able to call the constructors for its virtual base class, A. If B inherits A publicly, the above example would compile.

3.1.2.526 E2248: Cannot find default constructor to initialize array element of type 'class' (C++)

When declaring an array of a class that has constructors, you must either explicitly initialize every element of the array, or the class must have a default constructor.

The compiler will define a default constructor for a class unless you have defined any constructors for the class.

3.1.2.527 E2251: Cannot find default constructor to initialize base class 'class' (C++)

Whenever a C++ derived class 'class2' is constructed, each base class 'class1' must first be constructed.

If the constructor for 'class2' does not specify a constructor for 'class1' (as part of 'class2's' header), there must be a constructor class1::class1() for the base class.

This constructor without parameters is called the default constructor.

The compiler will supply a default constructor automatically unless you have defined any constructor for class 'class1'.

In that case, the compiler will not supply the default constructor automatically--you must supply one.

```
class Base {
public:
   Base(int) {}
};
class Derived = public Base {
   Derived():Base(1) {}
}
// must explicitly call the Base constructor, or provide a
// default constructor in Base.
```

Class members with constructors must be initialized in the class' initializer list, for example:

```
class A {
public
   A( int );
};
class B {
public:
   A a;
   B() : a( 3 ) {}; //ok
};
```

3.1.2.528 E2252: 'catch' expected (C++)

In a C++ program, a 'try' block must be followed by at least one 'catch' block.

3.1.2.529 E2253: Calling convention must be attributed to the function type, not the closure (C++)

The calling convention is in the wrong place in the closure declaration. For example,

```
int __fastcall (__closure * x)()
```

will compile, but

```
int (__fastcall __closure * x)()
```

will not.

3.1.2.530 E2261: Use of dispid with a property requires a getter or setter (C++)

This property needs either a getter or a setter.

3.1.2.531 E2262: '__except' or '__finally' expected following '__try' (C++)

In C, a '__try block' must be followed by a '__except' or '__finally' handler block.

3.1.2.532 E2270: > expected (C++)

A new-style cast (for example, dynamic_cast) was found with a missing closing ">".

3.1.2.533 E2273: 'main' cannot be declared as static or inline (C++)

You cannot make main static or inline. For example, you cannot use static int main() or inline int main().

3.1.2.534 E2281: Identifier1 requires definition of Identifier2 as a pointer type (C++)

To use Identifier1, there needs to be a definition for Identifier2, which is a type.

Example where __classid is the first identifier and TClass, which can be found in clx.h, is the second one:

```
// #include <clx/clx.h> missing
struct __declspec(delphiclass)bar
{
     virtual int barbara(void);
};
void *foo(void)
{
     return classid(bar);    // Error
}
```

3.1.2.535 E2289: __published or __automated sections only supported for Delphi classes (C++)

The compiler needs to generate a special kind of vtable for classes containing __published and __automated sections. Therefore, these sections are only supported for Delphi style classes.

Example

```
structregclass
{
int mem;
__published:// Error: no Delphi style class
int __property ip = { read = mem, write = mem };
```

```
};
struct__declspec(delphiclass) clxclass
{
int mem;
__published:// OK
int __property ip = { read = mem, write = mem };
};
```

3.1.2.536 E2298: Cannot generate 'function' from template function 'template' (C++)

A call to a template function was found, but a matching template function cannot be generated from the function template.

3.1.2.537 E2301: Cannot use templates in closure arguments -- use a typedef (C++)

When declaring a closure type, the arguments passed to that closure must be of a simple type. Templates are not accepted. To pass a reference to an object of template type to a closure, you must declare a typedef, which counts as a simple type name.

Example

```
typedef my_class<int> mci;
typedef void (__closure * func) (const mci& object);
```

3.1.2.538 E2307: Type 'type' is not a defined class with virtual functions (C++)

A dynamic_cast was used with a pointer to a class type that is either undefined, or doesn't have any virtual member functions.

3.1.2.539 E2315: 'Member' is not a member of 'class', because the type is not yet defined (C++)

The member is being referenced while the class has not been fully defined yet. This can happen if you forward declare class X, declare a pointer variable to X, and reference a member through that pointer; for example:

```
class X;
X * oneX;
int test() { return oneX->i; }
```

3.1.2.540 E2318: 'type' is not a polymorphic class type (C++)

This error is generated if the -RT compiler option (for runtime type information) is disabled and either

dynamic_cast was used with a pointer to a class

or

you tried to delete a pointer to an object of a class that has a virtual destructor

3.1.2.541 E2323: Illegal number suffix (C++)

A numeric literal is followed by a suffix that is not recognized by the compiler.

Example

```
int i = 1234i15;   // Error: no i15 suffix
int j = 1234i16;   // OK
```

3.1.2.542 E2326: Use __declspec(spec1[, spec2]) to combine multiple __declspecs (C++)

When you want to use several __declspec modifiers, the compiler will complain if you don't combine them into one __declspec. For example:

```
int __declspec(__import) __declspec(delphiclass) X
```

will give an error. Use the following instead:

```
int __declspec(__import, delphiclass) X
```

3.1.2.543 E2328: Classes with properties cannot be copied by value (C++)

This error occurs if you attempt to use the default assignment operator. For example, the following code generates this error given two labels on a form:

```
*Label1->Font = *Label2->Font;
```

3.1.2.544 E2331: Number of allowable option contexts exceeded (C++)

You have interspersed too many source-code option changes (using #pragma option) between template declarations. For example:

```
#pragma option -x
template<class T> class foo1 { };
#pragma option -a3
template<class T> class foo2 { };
#pragma option -b
template<class T> class foo3 { };
#pragma option -k-
```

You need to break your source code into smaller files.

3.1.2.545 E2332: Variable 'variable' has been optimized and is not available (C++)

You have tried to inspect, watch, or otherwise access a variable which the optimizer removed.

This variable is never assigned a value and has no stack location.

3.1.2.546 E2476: Cannot overload 'function' (C++)

You cannot overload the specified function. This error is displayed if you tried to declare a function with the same name as another function, but the redeclaration is not legal. For example, if both functions have the 'extern "C"' linkage type, only one 'extern "C"' function can have a given name.

3.1.2.547 E2346: 'x' access specifier of property 'property' must be a member function (C++)

Only member functions or data members are allowed in access specifications of properties.

Example

```
int GlobalGetter(void)
{
     return 0;
}
struct pbase
{
     intMemberGetter(void) {return 1;}
     int __property ip1 = { read = GlobalGetter }; // Error
     int __property ip2 = { read = MemberGetter }; // OK
};
```

3.1.2.548 E2347: Parameter mismatch in access specifier 'specifier' of property 'property' (C++)

The parameters of the member function used to access a property don't match the expected parameters.

Example

```
structpbase
{
       voidSetter1(void){}
       voidSetter2(int){}
     int __property ip1 = { write = Setter1 }; // Error
     int __property ip2 = { write = Setter2 }; // OK
};
```

3.1.2.549 E2348: Storage specifier not allowed for array properties (C++)

Array properties cannot have a storage specification.

Example

```
struct pbase
{
    int __property ap[char *] =
                           { stored = false }; // Error
};
```

3.1.2.550 E2351: Static data members not allowed in __published or __automated sections (C++)

Only nonstatic data members and member functions are allowed in __published or __automated sections.

Example

```
struct__declspec(delphiclass) clxclass
{
__published:
static intstaticDataMember;// Error
};
```

3.1.2.551 E2353: Class 'classname' is abstract because of 'member = 0' (C++)

This message is issued immediately after the "Cannot create instance of abstract class 'classname' error message and is intended to make it easier to figure out why a particular class is considered abstract by the compiler.

For example, consider the following example of an illegal attempt to instantiate an abstract class:

```
struct VB
{
virtualvoid  f() = 0;
virtualvoid  g() = 0;
virtualvoid  h() = 0;
};
struct D1 : virtual VB
{
void   f();
};
struct D2 : virtual VB
{
void   h();
};
struct DD : D1, D2
{
}
v;   // error 'DD' is an abstract class
```

The above code will cause the following two error messages:

```
Error TEST.CPP 21: Cannot create instance of abstract class 'DD'
Error TEST.CPP 21: Class 'DD' is abstract because of 'VB::g() = 0'
```

3.1.2.552 E2359: Reference member 'member' initialized with a non-reference parameter (C++)

An attempt has been made to bind a reference member to a constructor parameter. Since the parameter will cease to exist the moment the constructor returns to its caller, this will not work correctly.

3.1.2.553 E2369: Cannot use the result of a property assignment as an rvalue' (C++)

The result of a property assignment is an lvalue. This implies for instance that chained assignments of properties is not allowed; for example, x = y = 5, where both x and y are properties. Certain embedded assignments of properties can also produce errors; for example, x != (y = z), where y is a property.

3.1.2.554 E2370: Simple type name expected (C++)

To ensure interoperability between Delphi and C++, there are restrictions on the type names mentioned in the parameter lists of published closure types. The parameter types have to be simple type names with optional const modifier and pointer or reference notation.

So when declaring a closure type, the arguments passed to that closure must be of a simple type. For example, templates are not accepted. To pass a reference to an object of template type to a closure, you must declare a typedef, which counts as a simple type name.

Example

```
struct __declspec(delphiclass) foo
{
     typedef void __fastcall (__closure *foo1)(SomeTemplateType<int> *);
     typedef SomeTemplateType<int> SimpleTypeName;
     typedef void __fastcall (__closure *foo2)(SimpleTypeName *);
published:
     __property foo1 prop1;  // Error
     __property foo2 prop2;  // OK
};
```

3.1.2.555 E2398: Template function argument 'argument' not used in argument types (C++)

The given argument was not used in the argument list of the function.

The argument list of a template function must use all of the template formal arguments; otherwise, there is no way to generate a template function instance based on actual argument types.

3.1.2.556 E2419: Error while instantiating template 'template' (C++)

An error occurred during the instantiation of a particular template. This message always follows some other error message that indicates what actually went wrong. This message is displayed to help track down which template instantiation introduced the problem.

3.1.2.557 E2424: Template class nesting too deep: 'class' (C++)

The compiler imposes a certain limit on the level of template class nesting. This limit is usually only exceeded through a recursive template class dependency.

When this nesting limit is exceeded, the compiler issues this error message for all of the nested template classes. This usually

makes it easy to spot the recursion.

This error message is always followed by the fatal error "Out of memory".

3.1.2.558 E2457: Delphi style classes must be caught by reference (C++)

You can only catch a Delphi style object by pointer.

Example

```
void foo(TObject *p)
{
     try
     {
          throw(p);
     }
     catch    (TObject o)    // Error
     {
     }
     catch  (TObject *op)    // OK
     {
     }
}
```

3.1.2.559 E2458: Delphi classes have to be derived from Delphi classes (C++)

You cannot derive a Delphi style class from a non-Delphi style class.

Example

```
struct base// base not a Delphi style class
{
intbasemem;
};
struct __declspec(delphiclass) derived : base // or
{
intderivedmem;
};
```

3.1.2.560 E2459: Delphi style classes must be constructed using operator new (C++)

Delphi style classes cannot be statically defined. They have to be constructed on the heap.

Example

```
voidfoo(void)
{
     Tobject   o1;     // Error;
     Tobject  *o2 = new TObject();
}
```

3.1.2.561 E2460: Delphi style classes require exception handling to be enabled (C++)

If you are using Delphi style classes in your program, you cannot turn off exception handling (compiler option `-x-`) when compiling your source code.

3.1.2.562 E2463: 'base' is an indirect virtual base class of 'class' (C++)

You can't create a pointer to a C++ member of a virtual base class.

You have attempted to create such a pointer (either directly, or through a cast) and access an inaccessible member of one of your base classes.

3.1.2.563 Null pointer assignment (C++)

When a small or medium memory model program exits, a check is made to determine if the contents of the first few bytes within the program's data segment have changed. These bytes would never be altered by a working program. If they have been changed, this message is displayed to inform you that (most likely) a value was stored to an uninitialized pointer.

The program might appear to work properly in all other respects; however, this is a serious bug which should be attended to immediately. Failure to correct an uninitialized pointer can lead to unpredictable behavior (including locking the computer up in the large, compact, and huge memory models).

You can use the integrated debugger to track down null pointers.

3.1.2.564 E2268: Call to undefined function 'function' (C++)

Your source file declared the current function to return some type other than `void` in C++ (or `int` in C), but the compiler encountered a return with no value. All `int` functions are exempt in C because in old versions of C, there was no `void` type to indicate functions that return nothing.

3.1.2.565 E2375: Assembler stack overflow (C++)

The assembler ran out of memory during compilation. Review the portion of code flagged by the error message to ensure that it uses memory correctly.

3.1.2.566 Initializing enumeration with type (C++)

You're trying to initialize an enum variable to a different type. For example,

```
enum count { zero, one, two } x = 2;
```

will result in this warning, because 2 is of type int, not type enum count. It is better programming practice to use an enum identifier instead of a literal integer when assigning to or initializing enum types.

This is an error, but is reduced to a warning to give existing programs a chance to work.

3.1.2.567 <name> is not a valid identifier (C++)

The identifier name is invalid. Ensure that the first character is a letter or an underscore (_). The characters that follow must be letters, digits, or underscores, and there can not be any spaces in the identifier.

3.1.2.568 Example for "Temporary used ..." error messages (C++)

In this example, function f requires a reference to an int, and c is a char:

```
f(int&);
char c;
f(c);
```

Instead of calling f with the address of c, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

3.1.2.569 Application is running (C++)

The application you tried to run is already running.

For Windows, make sure the message loop of the program has properly terminated.

```
PostQuitMessage(0);
```

3.1.2.570 Printf/Scanf floating-point formats not linked (C++)

Floating-point formats contain formatting information that is used to manipulate floating-point numbers in certain runtime library functions, such as scanf() and atof(). Typically, you should avoid linking the floating-point formats (which take up about 1K) unless they are required by your application. However, you must explicitly link the floating-point formats for programs that manipulate fields in a limited and specific way.

Refer to the following list of potential causes (listed from most common to least common) to determine how to resolve this error:

- CAUSE: Floating point set to None. You set the floating-point option to None when it should be set to either Fast or Normal.
- FIX: Set Floating Point to Fast or Normal.
- CAUSE: Either the compiler is over-optimizing or the floating-point formats really do need to be linked. You need the floating-point formats if your program manipulates floats in a limited and specific way. Under certain conditions, the compiler will ignore floating-point usage in scanf(). For example, this may occur when trying to read data into a float variable that is part of an array contained in a structure.
- FIX: Add the following code to one source module:

```
extern _floatconvert;
#pragma extref _floatconvert
```

CAUSE: You forgot to put the address operator & on the scanf variable expression. For example:

```
float foo;
scanf("%f", foo);
```

FIX: Change the code so that the & operator is used where needed. For example, change the above code to the following:

```
float foo;
scanf("%f", &foo);
```

3.1.2.571 W8000: Ambiguous operators need parentheses (C++)

(Command-line option to display warning: -wamb)

This warning is displayed whenever two shift, relational, or bitwise-Boolean operators are used together without parentheses.

Also, an addition or subtraction operator that appears without parentheses with a shift operator will produce this warning.

3.1.2.572 W8060: Possibly incorrect assignment (C++)

(Command-line option to suppress warning: -w-pia)

This warning is generated when the compiler encounters an assignment operator as the main operator of a conditional expression (part of an if, while, or do-while statement).

This is usually a typographical error for the equality operator.

If you want to suppress this warning, enclose the assignment in parentheses and compare the whole thing to zero explicitly.

For example, this code

```
if (a = b) ...
```

should be rewritten as

```
if ((a = b) != 0) ...
```

3.1.2.573 W8002: Restarting compile using assembly (C++)

(Command-line option to suppress warning: -w-asc)

The compiler encountered an asm with no accompanying or `#pragma` inline statement.

The compile restarts using assembly language capabilities.

Default = On

3.1.2.574 W8003: Unknown assembler instruction (C++)

(Command-line option to suppress warning: -w-asm)

The compiler encountered an inline assembly statement with a disallowed opcode or an unknown token. Check the spelling of the opcode or token.

`Note:` You will get a separate error message from the assembler if you entered illegal assembler source code.

This warning is off by default.

3.1.2.575 W8052: Base initialization without a class name is now obsolete (C++)

(Command-line option to suppress warning: -w-obi)

Early versions of C++ provided for initialization of a base class by following the constructor header with just the base class

constructor parameter list.

It is now recommended to include the base class name.

This makes the code much clearer, and is required when you have multiple base classes.

Old way

```
derived::derived(int i) : (i, 10) { ... }
```

New way

```
derived::derived(int i) : base(i, 10) { ... }
```

3.1.2.576 E2117: Bit fields must be signed or unsigned int (C++)

(Command-line option to display warning: -wbbf)

In ANSI C, bit fields may not be of type signed char or unsigned char.

When you're not compiling in strict ANSI mode, the compiler allows these constructs, but flags them with this warning.

3.1.2.577 W8064: Call to function with no prototype (C++)

(Command-line option to suppress warning: -w-pro)

This message is given if the "Prototypes required" warning is enabled and you call a function without first giving a prototype for that function.

3.1.2.578 W8065: Call to function 'function' with no prototype (C++)

This message is given if the "Prototypes required" warning is enabled and you call function 'function' without first giving a prototype for that function.

3.1.2.579 W8009: Constant is long (C++)

(Command-line option to display warning: -wcln)

The compiler encountered one of the following:

- a decimal constant greater than 32,767 or
- an octal, hexadecimal, or decimal constant greater than 65,535 without a letter l or L following it

The constant is treated as a long.

3.1.2.580 W8008: Condition is always true OR W8008 Condition is always false (C++)

(Command-line option to suppress warning: -w-ccc)

Whenever the compiler encounters a constant comparison that (due to the nature of the value being compared) is always true or false, it issues this warning and evaluates the condition at compile time.

For example:

```
void proc(unsigned x){
  if (x >= 0)        /* always 'true' */
  {
     ...
  }
}
```

3.1.2.581 W8012: Comparing signed and unsigned values (C++)

(Command-line option to suppress warning: -w-csu)

Since the ranges of signed and unsigned types are different the result of an ordered comparison of an unsigned and a signed value might have an unexpected result.

Example

```
#pragma warn +csu
boolfoo(unsigned u, int i)
{
return u < i;
}
```

3.1.2.582 W8010: Continuation character \ found in // comment (C++)

(Command-line option to suppress warning: -w-com)

This warning message is issued when a C++ // comment is continued onto the next line with backslash line continuation.

The intention is to warn about cases where lines containing source code unintentionally become part of a comment because that comment happened to end in a backslash.

If you get this warning, check carefully whether you intend the line after the // comment to be part of the comment. If you don't, either remove the backslash or put some other character after it. If you do, it's probably better coding style to start the next comment line with // also.

The warning can be disabled altogether with #pragma warn -com.

3.1.2.583 W8080: 'identifier' is declared but never used (C++)

(Command-line option to display warning: -wuse)

The specified identifier was never used. This message can occur in the case of either local or static variables. It occurs when the source file declares the named local or static variable as part of the block just ending, but the variable was never used.

In the case of local variables, this warning occurs when the compiler encounters the closing brace of the compound statement or function. In the case of static variables, this warning occurs when the compiler encounters the end of the source file.

For example:

```
// Need to compile with -w to make this warning show up!
#pragma option -w
int foo()
{
        int x;
        return 0;
}
```

3.1.2.584 W8014: Declaration ignored (C++)

(Command-line option to suppress warning: -w-dig)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

Default = On

3.1.2.585 W8068: Constant out of range in comparison (C++)

(Command-line option to suppress warning: -w-rng)

Your source file includes a comparison involving a constant sub-expression that was outside the range allowed by the other sub-expression's type.

For example, comparing an unsigned quantity to -1 makes no sense.

To get an unsigned constant greater than 32,767 (in decimal), you should either

- cast the constant to unsigned--for example, (unsigned) 65535, or
- append a letter u or U to the constant--for example, 65535u.

Whenever this message is issued, the compiler still generates code to do the comparison.

If this code ends up always giving the same result (such as comparing a char expression to 4000), the code will still perform the test.

3.1.2.586 W8016: Array size for 'delete' ignored (C++)

(Command-line option to suppress warning: -w-dsz)

The C++ IDE issues this warning when you've specified the array size when deleting an array.

With the new C++ specification, you don't need to make this specification. The compiler ignores this construct.

This warning lets older code compile.

3.1.2.587 W8082: Division by zero (C++)

(Command-line option to suppress warning: -w-zdi)

A divide or remainder expression had a literal zero as a divisor.

3.1.2.588 W8018: Assigning 'type' to 'enumeration' (C++)

(Command-line option to suppress warning: -w-eas)

Assigning an integer value to an enum type.

This is an error in C++, but is reduced to a warning to give existing programs a chance to work.

3.1.2.589 W8006: Initializing 'identifier' with 'identifier' (C++)

(Command-line option to suppress warning: `-w-bei`)

You're trying to initialize an `enum` variable to a different type.

For example, the following initialization will result in this warning, because 2 is of type int, not type enum count:

```
enum count zero, one, two x = 2;
```

It is better programming practice to use an enum identifier instead of a literal integer when assigning to or initializing enum types.

This is an error, but is reduced to a warning to give existing programs a chance to work.

3.1.2.590 W8001: Superfluous & with function (C++)

(Command-line option to display warning: -wamp)

An address-of operator (&) is not needed with function name; any such operators are discarded.

3.1.2.591 W8020: 'identifier' is declared as both external and static (C++)

(Command-line option to suppress warning: -w-ext)

This identifier appeared in a declaration that implicitly or explicitly marked it as global or external, and also in a static declaration.

The identifier is taken as static.

You should review all declarations for this identifier.

3.1.2.592 W8007: Hexadecimal value contains more than three digits (C++)

(Command-line option to suppress warning = -w-big)

Under older versions of C, a hexadecimal escape sequence could contain no more than three digits.

The ANSI standard allows any number of digits to appear as long as the value fits in a byte.

This warning results when you have a long hexadecimal escape sequence with many leading zero digits (such as \x00045).

Older versions of C would interpret such a string differently.

3.1.2.593 W8024: Base class 'class1' is also a base class of 'class2' (C++)

(Command-line option to suppress warning: -w-ibc)

A class inherits from the same base class both directly and indirectly. It is best to avoid this non-portable construct in your program code.

3.1.2.594 W8022: 'function1' hides virtual function 'function2' (C++)

(Command-line option to suppress warning: -w-hid)

A virtual function in a base class is usually overridden by a declaration in a derived class.

In this case, a declaration with the same name but different argument types makes the virtual functions inaccessible to further derived classes.

3.1.2.595 W8023: Array variable 'identifier' is near (C++)

(Command-line option to suppress warning: -w-ias)

When you use set the Far Data Threshold option, the compiler automatically makes any global variables that are larger than the threshold size be far.

When the variable is an initialized array with an unspecified size, its total size is not known when the compiler must decide whether to make it near or far, so the compiler makes it near.

The compiler issues this warning if the number of initializers given for the array causes the total variable size to exceed the data size threshold.

If the fact that the compiler made the variable be near causes problems, make the offending variable explicitly far.

To do this, insert the keyword "far" immediately to the left of the variable name in its definition.

3.1.2.596 W8061: Initialization is only partially bracketed (C++)

(Command-line option to display warning: `-wpin`)

When structures are initialized, braces can be used to mark the initialization of each member of the structure. If a member itself is an array or structure, nested pairs of braces can be used. This ensures that the compiler's idea and your idea of what value goes with which member are the same. When some of the optional braces are omitted, the compiler issues this warning.

3.1.2.597 W8038: constant member 'identifier' is not initialized (C++)

(Command-line option to suppress warning: -w-nci)

This C++ class contains a constant member 'member' that doesn't have an initialization.

Note that constant members can be initialized only; they can't be assigned to.

3.1.2.598 W8071: Conversion may lose significant digits (C++)

(Command-line option to display warning: -wsig)

For an assignment operator or some other circumstance, your source file requires a conversion from a larger integral data type to a smaller integral data type where the conversion exists.

Because the integral data type variables don't have the same size, this kind of conversion might alter the behavior of a program.

3.1.2.599 W8043: Macro definition ignored (C++)

(Command-line option to suppress warning: -w-nma)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

3.1.2.600 W8017: Redefinition of 'x' is not identical (C++)

(Command-line option to suppress warning: -w-dup)

Your source file redefined the macro 'ident' using text that was not exactly the same as the first definition of the macro.

The new text replaces the old.

3.1.2.601 W8079: Mixing pointers to different 'char' types (C++)

(Command-line option to display warning: -wucp)

You converted a signed char pointer to an unsigned char pointer, or vice versa, without using an explicit cast. (Strictly speaking, this is incorrect, but it is often harmless.)

3.1.2.602 W8067: Both return and return with a value used (C++)

(Command-line option to suppress warning: -w-ret)

The current function has return statements with and without values.

This is legal C, but almost always generates an error.

Possibly a return statement was omitted from the end of the function.

3.1.2.603 W8048: Use qualified name to access member type 'identifier' (C++)

(Command-line option to suppress warning: -w-nst)

In previous versions of the C++ specification, `typedef` and tag names declared inside classes were directly visible in the global scope.

In the latest specification of C++, these names must be prefixed with class`::`qualifier if they are to be used outside of their class scope.

The compiler issues this warning whenever a name is uniquely defined in a single class. The compiler permits this usage without class`::`. This allows older versions of code to compile.

3.1.2.604 W8039: Constructor initializer list ignored (C++)

(Command-line option to suppress warning: -w-ncl)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

3.1.2.605 W8040: Function body ignored (C++)

(Command-line option to suppress warning: -w-nfd)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

3.1.2.606 W8042: Initializer for object 'x' ignored (C++)

(Command-line option to suppress warning: -w-nin)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

3.1.2.607 W8044: #undef directive ignored (C++)

(Command-line option to suppress warning: -w-nmu)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

3.1.2.608 W8037: Non-const function 'function' called for const object (C++)

(Command-line option to suppress warning = -w-ncf)

A non-const member function was called for a const object.

(This is an error, but was reduced to a warning to give existing programs a chance to work.)

3.1.2.609 W8051: Non-volatile function 'function' called for volatile object (C++)

(Command-line option to suppress warning: -w-nvf)

In C++, a class member function was called for a volatile object of the class type, but the function was not declared with volatile following the function header. Only a volatile member function can be called for a volatile object.

For example, if you have

```
class c
{
public:
  f() volatile;
  g();
};
volatile c vcvar;
```

it is legal to call vcvar.f(), but not to call vcvar.g().

3.1.2.610 W8019: Code has no effect (C++)

(Command-line option to suppress warning: -w-eff)

This warning is issued when the compiler encounters a statement with some operators that have no effect.

For example, the statement

```
a + b;
```

has no effect on either variable.

The operation is unnecessary and probably indicates a bug.

3.1.2.611 W8057: Parameter 'parameter' is never used (C++)

(Command-line option to suppress warning: -w-par)

The named parameter, declared in the function, was never used in the body of the function.

This might or might not be an error and is often caused by misspelling the parameter.

This warning can also occur if the identifier is redeclared as an automatic (local) variable in the body of the function.

The parameter is masked by the automatic variable and remains unused.

3.1.2.612 W8070: Function should return a value (C++)

(Command-line option to suppress warning: `-w-rvl`)

This function was declared (maybe implicitly) to return a value.

The compiler found a return statement without a return value, or it reached the end of the function without finding a return statement.

Either return a value or change the function declaration to return void.

3.1.2.613 W8047: Declaration of static function function ignored (C++)

(Command-line option to suppress warning: -w-nsf)

An error has occurred while using the command-line utility H2ASH. See the online file "tsm_util.txt" for further information about this utility.

3.1.2.614 W8041: Negating unsigned value (C++)

(Command-line option to suppress warning: -w-ngu)

Basically, it makes no sense to negate an unsigned value because the result will still be unsigned.

Example

```
#pragma warn +ngu
unsignedfoo(unsigned u)
{
return-u;
}
```

3.1.2.615 W8054: Style of function definition is now obsolete (C++)

(Command-line option to suppress warning = -w-ofp)

In C++, this old C style of function definition is illegal:

```
int func(p1, p2) int p1, p2; { /* ... */ }
```

This practice might not be allowed by other C++ compilers.

3.1.2.616 W8025: Ill-formed pragma (C++)

(Command-line option to suppress warning: -w-ill)

A pragma does not match one of the pragmas expected by the compiler.

3.1.2.617 W8063: Overloaded prefix operator 'operator' used as a postfix operator (C++)

(Command-line option to suppress warning: -w-pre)

The C++ specification allows you to overload both the prefix and postfix versions of the ++ and -- operators.

Whenever the prefix operator is overloaded, but is used in a postfix context, the compiler uses the prefix operator and issues this warning.

This allows older code to compile.

3.1.2.618 W8015: Declare 'type' prior to use in prototype (C++)

(Command-line option to suppress warning: -w-dpu)

When a function prototype refers to a structure type that has not previously been declared, the declaration inside the prototype is not the same as a declaration outside the prototype.

For example,

```
int func(struct s *ps);  struct s  /* ... */ ;
```

Because there is no "struct s" in scope at the prototype for func, the type of parameter ps is pointer to undefined struct s, and is not the same as the "struct s" that is later declared.

This will result in later warning and error messages about incompatible types, which would be very mysterious without this warning message.

To fix the problem, you can move the declaration for "struct s" ahead of any prototype that references it, or add the incomplete type declaration "struct s;" ahead of any prototype that references "structs".

If the function parameter is a struct, rather than a pointer to struct, the incomplete declaration is not sufficient.

You must then place the struct declaration ahead of the prototype.

3.1.2.619 W8069: Nonportable pointer conversion (C++)

(Command-line option to suppress warning: `-w-rpt`)

A nonzero integral value is used in a context where a pointer is needed or where an integral value is needed; the sizes of the integral type and pointer are the same.

Use an explicit cast if this is what you really meant to do.

3.1.2.620 W8066: Unreachable code (C++)

(Command-line option to suppress warning: -w-rch)

A break, continue, goto, or return statement was not followed by a label or the end of a loop or function.

The compiler checks while, do, and for loops with a constant test condition, and attempts to recognize loops that can't fall through.

3.1.2.621 W8029: Temporary used for parameter '???' (C++)

(Command-line option to suppress warning: -w-lvc)

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter.

The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

In the following example, function f requires a reference to an int, and c is a char:

```
f(int &amp);
char c;
f(c);
```

Instead of calling f with the address of c, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

3.1.2.622 W8031: Temporary used for parameter 'parameter' OR W8029 Temporary used for parameter 'number' OR W8030 Temporary used for parameter 'parameter' in call to 'function' OR W8032 Temporary used for parameter 'number' in call to 'function' (C++)

(Command-line option to suppress warning: -w-lvc)

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type.

If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter.

The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

3.1.2.623 W8032: Temporary used for parameter 2 in call to '???' (C++)

(Command-line option to suppress warning: -w-lvc)

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter.

The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

In the following example, function f requires a reference to an int, and c is a char:

```
f(int &amp);
char c;
f(c);
```

Instead of calling f with the address of c, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

3.1.2.624 W8028: Temporary used to initialize 'identifier' (C++)

(Command-line option to suppress warning: -w-lin)

In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type.

If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter.

The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.

Example

In this example, function f requires a reference to an int, and c is a char:

```
f(int&);
char c;
f(c);
```

Instead of calling f with the address of c, the compiler generates code equivalent to the C++ source code:

```
int X = c, f(X);
```

3.1.2.625 W8074: Structure passed by value (C++)

(Command-line option to display warning: -wstv)

This warning is generated any time a structure is passed by value as an argument.

It is a frequent programming mistake to leave an address-of operator (&) off a structure when passing it as an argument.

Because structures can be passed by value, this omission is acceptable.

This warning provides a way for the compiler to warn you of this mistake.

3.1.2.626 W8011: Nonportable pointer comparison (C++)

(Command-line option to suppress warning: -w-cpt)

Your source file compared a pointer to a non-pointer other than the constant 0.

You should use a cast to suppress this warning if the comparison is proper.

3.1.2.627 W8075: Suspicious pointer conversion (C++)

(Command-line option to suppress warning: -w-sus)

The compiler encountered some conversion of a pointer that caused the pointer to point to a different type.

You should use a cast to suppress this warning if the conversion is proper.

A common cause of this warning is when the C compiler converts a function pointer of one type to another (the C++ compiler generates an error when asked to do that). It can be suppressed by doing a typecast. Here is a common occurrence of it for Windows programmers:

```
#define STRICT
#include <windows.h>
LPARAM _export WndProc( HWND , UINT , WPARAM , LPARAM );
test() {
   WNDCLASS wc;
   wc.lpfnWndProc = WndProc;  //warning
}
```

It is suppressed by making the assignment to lpfnWndProc as follows:

```
wc.lpfnWndProc = ( WNDPROC ) WndProc;
```

3.1.2.628 W8059: Structure packing size has changed (C++)

(Command-line option to suppress warning: -w-pck)

This warning message is issued when the structure alignment is different after including a file than it was before including that file.

The intention is to warn you about cases where an include file changes structure packing, but by mistake doesn't restore the original setting at the end. If this is intentional, you can give a #pragma nopackwarning directive at the end of an include file to disable the warning for this file.

The warning can be disabled altogether by #pragma warn -pck.

3.1.2.629 W8045: No declaration for function 'function' (C++)

(Command-line option to display warning: -wnod)

This message is given if you call a function without first declaring that function.

In C, you can declare a function without presenting a prototype, as in

```
int func();
```

In C++, every function declaration is also a prototype; this example is equivalent to

```
int func(void);
```

The declaration can be either classic or modern (prototype) style.

3.1.2.630 W8073: Undefined structure 'structure' (C++)

(Command-line option to display warning = `-wstu`)

Your source file used the named structure on some line before where the error is indicated (probably on a pointer to a structure) but had no definition for the structure.

This is probably caused by a misspelled structure name or a missing declaration.

3.1.2.631 W8013: Possible use of 'identifier' before definition (C++)

(Command-line option to display warning: -wdef)

Your source file used the variable 'identifier' in an expression before it was assigned a value.

The compiler uses a simple scan of the program to determine this condition.

If the use of a variable occurs physically before any assignment, this warning will be generated.

Of course, the actual flow of the program can assign the value before the program uses it.

3.1.2.632 W8004: 'identifier' is assigned a value that is never used (C++)

(Command-line option to suppress warning: -w-aus)

The variable appears in an assignment, but is never used anywhere else in the function just ending.

The warning is indicated only when the compiler encounters the closing brace.

The `#pragma warn -aus` switch has function-level granularity. You cannot turn off this warning for individual variables within a function; it is either off or on for the whole function.

3.1.2.633 W8081: Void functions may not return a value (C++)

(Command-line option to suppress warning: -w-voi)

Your source file declared the current function as returning void, but the compiler encountered a return statement with a value. The value of the return statement will be ignored.

Example

```
// This HAS to be in a "C" file.  In a "C++" file this would be an error
void foo()
{
        return 0;       // Can't return a value from a void function
}
```

3.1.2.634 W8078: Throw expression violates exception specification (C++)

(Command-line option to suppress warning: -w-thr)

This warning happens when you add an exception specification to a function definition and you throw a type in your function body that is not mentioned in your exception specification.

The following program would generate this warning:

```
int foo() throw(char*)  // I promise to only throw char*s
{
    throw 5;  // Oops, I threw an integer
    return 0;
}
```

3.1.2.635 W8021: Handler for 'type1' hidden by previous handler for 'type2' (C++)

(Command-line option to suppress warning: -w-hch)

This warning is issued when a handler for a type 'D' that is derived from type 'B' is specified after a handler for B', since the handler for 'D' will never be invoked.

3.1.2.636 W8056: Integer arithmetic overflow (C++)

The compiler detected an overflow condition in an integer math expression.

For example:

```
int X = 0xFFFF * 0xFFFF;
```

3.1.2.637 W8035: User-defined message (C++)

The error message for which you have requested Help is a user-defined warning.

In C++ code, user-defined messages are introduced by using the `#pragma` message compiler syntax.

`Note:` In addition to messages that you introduce with the `#pragma` message compiler syntax, user-defined warnings can be introduced by third party libraries. Should you require Help about a third party warning, please contact the vendor of the header file that issued the warning.

3.1.2.638 W8049: Use '> >' for nested templates Instead of '>>' (C++)

(Command-line option to suppress warning: -w-ntd)

Whitespace is required to separate the closing ">" in a nested template name, but since it is an extremely common mistake to leave out the space, the compiler accepts a ">>" with this warning.

3.1.2.639 W8026: Functions with exception specifications are not expanded inline (C++)

Also:Functions taking class by value arguments are not expanded inline

(Command-line option to suppress warning: -w-inl)

Exception specifications are not expanded inline: Check your inline code for lines containing exception specification.

Functions taking class-by-value argument(s) are not expanded inline: When exception handling is enabled, functions that take class arguments by value cannot be expanded inline.

`Note:` Functions taking class parameters by reference are not subject to this restriction.

3.1.2.640 W8058: Cannot create pre-compiled header: 'reason' (C++)

(Command-line option to suppress warning: -w-pch)

This warning is issued when pre-compiled headers are enabled but the compiler could not generate one, for one of the following reasons:

ReasonExplanation

write failedThe compiler could not write to the pre-compiled header file. This occurs if you specified an invalid location to cache precompiled headers or if the disk is full.

code in headerOne of the headers contained a non-inline function body.

initialized data in headerOne of the headers contained a global variable definition (in C, a global variable with an initializer; in C++ any variable not declared as 'extern').

header incompleteThe pre-compiled header ended in the middle of a declaration, for example, inside a class definition (this often happens when there is a missing closing brace in a header file).

3.1.2.641 W8046: Pragma option pop with no matching option push (C++)

The compiler encountered a #pragma option pop before a previous #pragma option push, or in the case of nesting, there are more occurrences of #pragma option pop than of #pragma option push.

3.1.2.642 W8050: No type OBJ file present; disabling external types option. (C++)

(Command-line option to suppress warning: -w-nto)

A precompiled header file references a type object file, but the type object file cannot be found. This is not a fatal problem but will make your object files larger than necessary.

3.1.2.643 W8027: Functions containing 'statement' are not expanded inline (C++)

(Command-line option to suppress warning: -w-inl)

Where:

'statement' can be any of the following:

- Static variables
- Aggregate initializers
- Some return statements
- Local destructors
- Some if statements
- Local classes
- Missing return statements
- Disallowed reserved words listed under "Reserved words" below.

Reserved words

Functions containing any of these reserved words can't be expanded inline, even when specified as inline:

asm	except
break	finally
case	for
continue	goto
defaults	switch
do	while

The function is still perfectly legal, but will be treated as an ordinary static (not global) function.

A copy of the function will appear in each compilation unit where it is called.

Description

If an inline function becomes too complex, the compiler is unable to expand it inline. However, because the function is so complex, expanding it inline is unlikely to provide significant performance enhancements.

For local destructors

You've created an inline function for which the compiler turns off inlining. You can ignore this warning; the function will be generated out of line.

3.1.2.644 W8036: Non-ANSI keyword used: 'keyword' (C++)

(Command-line option to display warning: -wnak)

A non-ANSI keyword (such as '__fastcall') was used when strict ANSI conformance was requested via the -A option.

3.1.2.645 W8053: 'ident' is obsolete (C++)

(Command-line option to suppress warning: -w-obs)

Issues a warning upon usage for any "C" linkage function that has been specified. This will warn about functions that are "obsolete".

Here's an example of it's usage:

```
#ifdef __cplusplus
extern "C" {
#endif
void my_func(void);
#ifdef __cplusplus
}
#endif
#pragma obsolete my_func
main()
{
my_func();   // Generates warning about obsolete function
}
```

3.1.2.646 W8103: Path 'path' and filename 'filename' exceed maximum size of 'n' (C++)

(Command-line option to display warning: -wstv)

In looking up include files, the C++ compiler has encountered a file whose path and filename contain more characters than are allowed in the Windows maximum. Rename the paths and filenames that you can, and shorten their names wherever possible.

3.1.2.647 W8062: Previous options and warnings not restored (C++)

The compiler didn't encounter a #pragma option pop after a previous #pragma option push, or in the case of nesting, there are more occurrences of #pragma option push than of #pragma option pop.

3.1.2.648 W8055: Possible overflow in shift operation (C++)

The compiler detects cases where the number of bits shifted over is larger than the number of bits in the affected variable; for example:

```
char c;  c >> 16;
```

3.1.2.649 W8072: Suspicious pointer arithmetic (C++)

This message indicates an unintended side effect to the pointer arithmetic (or array indexing) found in an expression.

Example

```
#pragma warn +spa
intarray[10];
intfoo(__int64 index)
{
returnarray[index];
}
```

The value of index is 64 bits wide while the address of array is only 32 bits wide.

3.1.2.650 W8033: Conversion to 'type' will fail for members of virtual base 'class' (C++)

(Command-line option to suppress warning: -w-mpc)

This warning is issued only if the `-Vv` option is in use.

The warning may be issued when a member pointer to one type is cast to a member pointer of another type and the class of the converted member pointer has virtual bases.

Encountering this warning means that at runtime, if the member pointer conversion cannot be completed, the result of the cast will be a NULL member pointer.

3.1.2.651 W8034: Maximum precision used for member pointer type 'type' (C++)

(Command-line option to suppress warning: -w-mpd)

When a member pointer type is declared, its class has not been fully defined, and the `-Vmd` option has been used, the compiler has to use the most general (and the least efficient) representation for that member pointer type. This can cause less efficient code to be generated (and make the member pointer type unnecessarily large), and can also cause problems with separate compilation; see the `-Vm` compiler switch for details.

3.1.2.652 E2537: Cannot create instance of abstract class (C++)

This class is an abstract class, which you cannot instantiate.

3.1.2.653 E2018: Cannot catch 'identifier' -- ambiguous base class 'identifier' (C++)

It is not legal to catch a class that contains more than one copy of a (non-virtual) base class. However, you can catch the exception as a more derived type. For example:

```
struct awkward : std::runtime_error, std::logic_error {};

 try {
 }
 catch( std::exception & ) { // ambiguous, not caught here
 }
 catch( std::runtime_error & ) { // caught here
 }
```

Note that the compiler might warn you that the second catch cannot be triggered because the first clause should catch all matching exceptions. But the ambiguous base case is caught by the second clause.

3.1.2.654 E2550: No arguments can follow a variadic template in an argument list (C++)

In an argument list, the variadic template must not be followed by arguments.

3.1.2.655 E2538: Static assert failed: '%s' (C++)

See Static Assertions (see page 498) for details about how to use static_assert., which is one of the C++0x features.

3.1.2.656 E2548: ... was unexpected; expression is not a variadic template pattern (C++)

This message pertains to the expected syntax of variadic templates.

3.1.2.657 E2543: Combination of options 'options' is not permitted (C++)

Eliminate the combination of options.

For example, you cannot combine the C++ compiler options -b- and -bi because -b- turns off all -b options, including -bi.

3.1.2.658 E2549: Operand is not a parameter pack (C++)

A parameter pack is required in the indicated location. This message pertains to variadic templates.

3.1.2.659 E2544: Function exception specifications do not match (C++)

The throw specifications on two function declaration/definitions which refer to the same function do not agree. Rewrite the function exception specifications to match.

3.1.2.660 E2536: Incomplete type cannot be part of a exception declaration (C++)

This error occurs when an incomplete type is used in exception declaration. From C++ ISO 14882:1998 15.3.1: "The exception-declaration shall not denote an incomplete type. The exception-declaration shall not denote a pointer or reference to an incomplete type, other than void*, const void*, volatile void*, or const volatile void*. Types shall not be defined in an exception-declaration.

3.1.2.661 E2535: Incomplete type cannot be part of a throw specification (C++)

This error occurs when an incomplete type is used in a throw specification. From C++ ISO 14882:1998 15.1.3: "The type of the throw-expression shall not be an incomplete type, or a pointer or reference to an incomplete type, other than void*, const void*, volatile void*, or const volatile void*." This includes exception specifications (15.4.1).

3.1.2.662 E2532: Constant in new expression requires an initializer (C++)

If the type is const then either an initializer must be present or the non-POD class must have a user-declared constructor. Error in ANSI mode, warning if ANSI mode is not set.

3.1.2.663 E2541: Attribute '%s' cannot be set (C++)

This error occurs if the specified attribute cannot be applied to this entity.

3.1.2.664 E2545: Enum underlying type must be an integral (C++)

Ensure that the underlying type of the enumeration is an integral type.

The underlying type of a scoped enum is fixed to int unless explicitly provided by the user. The underlying type of a 'classic' enum remains unspecified unless fixed by the user.

For more information, see Strongly Typed Enums (see page 498).

3.1.2.665 E2546: Redeclaration of enum is not identical (C++)

This error occurs if:

- a re-declaration of an enum has a different underlying type
- a scoped enum is re-declared as an unscoped enum
- an unscoped enum is re-declared as a scoped enum

For more information, see Strongly Typed Enums (see page 498).

3.1.2.666 E2533: Parameter mismatch (wanted typename) (C++)

This error occurs when a parameter passed to a type trait function is not the correct type.

3.1.2.667 E2534: Integral constant expression expected (C++)

Integral constant expressions are described in section 5.19 of the C++ standard, and are sometimes referred to as "compile time constants". A working draft for the C++ standard can be found at

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf.

3.1.2.668 E2531: Parameter is an incomplete type (C++)

This error occurs when an incomplete type is passed into a type trait function that does not accept incomplete type parameters. For example:

```
class A;
void foo()
{
  __is_abstract(A); // error
}
```

3.1.2.669 E2539: Constant expression expected (C++)

You need to supply a constant expression in the indicated location.

For details about using a constant expression in a static assertion, see Static Assertions (see page 498).

3.1.2.670 E2547: ... expected (C++)

An ellipsis (...) is expected in the location indicated.

3.1.2.671 E2540: String literal expected (C++)

You need to specify a string literal in the position indicated in the error message.

A string literal is the second argument in a static assertion. For details about using a static assertion, see Static Assertions (see page 498).

3.1.2.672 E2552: This feature is not (yet) supported (C++)

Support might be added in a coming release.

3.1.2.673 E2542: '%s' is marked 'final' and cannot be overriden (C++)

The attribute 'final' applies to class definitions and to virtual member functions being declared in a class definition. If the attribute is specified for a class definition, it is equivalent to being specified for each virtual member function of that class, including inherited member functions. If a virtual member function f in some class B is marked 'final' and in a class D derived from B, a function D::f overrides B::f, the program is ill-formed.

3.1.2.674 E2553: %s mismatch in redeclaration of '%s' (C++)

Attributes specified on a declaration have to be a subset of attributes specified on a definition.

3.1.2.675 E2551: Return statement not allowed in __finally block (C++)

Rewrite the `__finally` block so that it does not contain a return statement.

For more information, see Writing a finally Block in C++ (see page 2023) and finally keyword (see page 550).

3.1.2.676 W8104: Local Static with constructor dangerous for multi-threaded apps (C++)

(Command-line option to suppress warning: -w-mls)

This warning is generated for local static objects with constructors for multithreaded programs. This situation can cause problems if two threads enter the containing function at the same time and no critical sections are present, allowing the constructor to potentially be called more than once.

3.1.2.677 W8106: %s are deprecated (C++)

You have used an old style syntax or language use that is deprecated -- that is, no longer recommended and might be phased out in the future.

3.1.2.678 W8110: Duplicate '%s' attribute directive ignored (C++)

(Command-line option to suppress warning: -w-dat)

This warning is generated if the same attribute was specified more than once for the same entity.

3.1.2.679 W8108: Constant in new expression requires an initializer (C++)

(Command-line option to suppress warning: -w-nci)

If the type is const then either an initializer must be present or the non-POD class must have a user-declared constructor. Error in ANSI mode, warning if ANSI mode is not set.

3.1.2.680 W8113: Inline function was declared with 'extern template' (C++)

(Command-line option to suppress warning: -w-iex)

'extern template' has no normative effect on inline functions. Implementations are encouraged to suppress out-of-line copies of inline functions that were declared with 'extern template'.

3.1.2.681 W8109: Parameter '%s' is a dependent type (C++)

(Command-line option to enable warning: -wpad)

This warning is generated if a parameter of an intrinsic function is a dependent type and the dependency cannot be resolved.

3.1.2.682 W8105: Reference/Constant member 'identifier' in class without constructors (C++)

(Command-line option to suppress warning: -w-mnc)

This warning is generated for a reference or constant member in a class without constructors if ANSI mode is not set. In ANSI mode this is an error.

3.1.2.683 W8107: Type name expected (C++)

(Command-line option to enable warning: -wntn)

This warning is generated if the type of a class/union member is omitted in its declaration and ANSI mode is not set. Always an error in ANSI mode.

3.1.2.684 W8112: Unresolved dependencies in expression (C++)

(Command-line option to suppress warning: -w-dex)

This warning is generated if dependencies in the expression cannot be resolved.

3.1.3 C++ Language Guide

This sections contains C++ language topics.

Topics

Name	Description
C++ Specifics (see page 385)	This section contains C++ Specific topics.
Keywords, Alphabetical Listing (see page 513)	This section contains Keywords, Alphabetical Listing topics.
Keywords, By Category (see page 579)	This section contains Keywords, By Category topics.
Language Structure (see page 586)	The topics in this section provide a formal definition of C++ language and its implementation in the CodeGear C++ compiler. They describe the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.
Lexical Elements (see page 660)	This section contains Lexical Element topics.
The Preprocessor (see page 687)	This section contains Preprocessor topics.

3.1.3.1 C++ Specifics

This section contains C++ Specific topics.

Topics

Name	Description
Classes (see page 386)	This section contains Class topics.
Class Templates Overview (see page 411)	This section contains Class Templates Overview topics.
Compiler Template Switches (see page 413)	This section contains Compiler Template Switch topics.
Constructors And Destructors (see page 415)	This section contains Constructor and Destructor topics.
C++ namespaces (see page 426)	This section contains C++ name space topics.
C++ Scope (see page 429)	This section contains C++ Scope topics.
Exporting And Importing Templates (see page 432)	This section contains Exporting And Importing Template topics.
Function Templates Overview (see page 433)	This section contains Function Templates Overview topics.
The new And delete Operators (see page 434)	This section contains new And delete Operator topics.
New-style Typecasting Overview (see page 438)	This section contains New-style Typecasting Overview topics.
Operator Overloading Overview (see page 438)	This section contains Operator Overloading Overview topics.
Overloading Operator Functions Overview (see page 441)	This section contains Overloading Operator Functions Overview topics.
Polymorphic Classes (see page 446)	This section contains Polymorphic Class topics.
Referencing (see page 451)	This section contains Typeid Operator topics.
Run-time Type Identification (RTTI) (see page 453)	This section contains Run-time Type Identification (RTTI) topics.
The Scope Resolution Operator (see page 454)	This section contains Scope Resolution Operator topics.
Stricter C++ Compiler (C++Builder 2007) (see page 454)	To more closely obey the rules of the C++ ANSI Standard, the C++ compiler shipping with C++Builder 2007 is stricter than previous versions. Code that did not generate errors in earlier versions of C++Builder might fail to compile beginning with C++Builder 2007. This section lists some of the common areas where the compiler is stricter. Each case is illustrated with an example showing the problem and how to update the code to compile with C++Builder 2007. Note that there are often many ways to bring offending code up to date. The appropriate method depends on the intent of the original... more (see page 454)
Templates (see page 460)	This section contains Template topics.
C++0x Features (C++Builder 2009) (see page 462)	C++Builder 2009 implements a number of the new features proposed in the C++0x standard. This section lists and describes these C++0x features. You can view the Working Paper draft that was used to guide the implementation of these features at Draft Working Paper. You can view a summary of the working group papers at Summary of Working Group Papers.
Unicode for C++ (see page 500)	This section contains Unicode related feature topics for C++.
Handling Delphi Features in C++Builder 2009 (see page 507)	This section contains topics describing how you can handle issues that might arise when C++Builder 2009 interacts with certain new Delphi features.

3.1.3.1.1 Classes

This section contains Class topics.

Topics

Name	Description
C++ Classes (see page 387)	C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that inherit the members of one or more base (or parent) classes. In C++, structures and unions are considered classes with certain access defaults. A simplified, “first-look” syntax for class declarations is class-key <type-info> class-name <: base-list> { <member-list> }; class-key is one of **class**, **struct**, or **union**. The optional type-info indicates a request for runtime type information about the... more (see page 387)
Friends Of Classes (see page 388)	This section contains Friends Of Class topics.

Class Member List (see page 390)	The optional member-list is a sequence including, but not exclusive to: • Data declarations (of any type, including enumerations, bit fields and other classes) • Nested type declarations • Nested type definitions • Template declarations • Template definitions • Function declarations • Function definitions • Constructors • A destructor Members of a class can optional have storage class specifiers and access modifiers. The objects thus defined are called class members. The storage class specifiers **auto**, **extern**, and **register** are not allowed. Members can be declared with the **static** storage class specifiers.
Inline Functions (see page 391)	This section contains Inline Function topics.
Class Name Scope (see page 393)	The scope of a class name is local. There are some special requirements if the class name appears more than once in the same scope. Class name scope starts at the point of declaration and ends with the enclosing block. A class name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can be referred to only by using the elaborated type specifier. This means that the class key, **class... more (see page 393)**
The Keyword This (see page 394)	This section contains Keyword this topics.
Class Names (see page 396)	**class-name** is any identifier unique within its scope. With structures, classes, and unions, class-name can be omitted. See Untagged structures and typedefs for discussion of untagged structures.
Member Scope (see page 396)	This section contains Member Scope topics.
Class Objects (see page 402)	Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including definition of member and friend functions and the redefinition of standard functions and operators when used with objects of a certain class. Redefined functions and operators are said to be overloaded. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called member functions for that class. C++ offers the overloading mechanism that allows... more (see page 402)
Virtual Base Classes (see page 403)	This section contains Virtual Base Class topics.
Class Types (see page 404)	The declaration creates a unique type, class type class-name. This lets you declare further class objects (or instances) of this type, and objects derived from this type (such as pointers to, references to, arrays of class-name, and so on):
Member Functions (see page 405)	A function declared without the **friend** specifier is known as a member function of the class. Functions declared with the **friend** modifier are called friend functions. Member functions are often referred to as methods in Delphi documentation. The same name can be used to denote more than one function, provided they differ in argument type or number of arguments.
VCL Class Declarations (see page 405)	**Syntax**
Class Methods (see page 406)	In C++Builder, a **class method** is a method that can be invoked on a class name, as well as on an instance of that class. In contrast, object methods can be invoked only on objects -- instances of a class.
Static Properties (see page 410)	Static properties have been implemented in C++Builder 2009 to enhance compatibility with the Delphi language

3.1.3.1.1.1 C++ Classes

C++ classes offer extensions to the predefined type system. Each class type represents a unique set of objects and the

operations (methods) and conversions available to create, manipulate, and destroy such objects. Derived classes can be declared that inherit the members of one or more base (or parent) classes.

In C++, structures and unions are considered classes with certain access defaults.

A simplified, “first-look” syntax for class declarations is

class-key <type-info> class-name

<: base-list> { <member-list> };

class-key is one of **class**, **struct**, or **union**.

The optional type-info indicates a request for runtime type information about the class. You can compile with the **–RT** compiler option, or you can use the **__rtti** keyword.

The optional base-list lists the base class or classes from which the class class-name will derive (or inherit) objects and methods. If any base classes are specified, the class class-name is called a derived class. The base-list has default and optional overriding access specifiers that can modify the access rights of the derived class to members of the base classes.

The optional member-list declares the class members (data and functions) of class-name with default and optional overriding access specifiers that can affect which functions can access which members.

See Also

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.2 Friends Of Classes

This section contains Friends Of Class topics.

Topics

Name	Description
Friends Of Classes (see page 389)	A **friend** F of a class X is a function or class, although not a member function of X, with full access rights to the private and protected members of X. In all other respects, F is a normal function with respect to scope, declarations, and definitions. Since F is not a member of X, it is not in the scope of X, and it cannot be called with the x.F and xptr->F selector operators (where x is an X object and xptr is a pointer to an X object). If the specifier **friend** is used with a function declaration... more (see page 389)

3.1.3.1.1.2.1 Friends Of Classes

A **friend** F of a class X is a function or class, although not a member function of X, with full access rights to the private and protected members of X. In all other respects, F is a normal function with respect to scope, declarations, and definitions.

Since F is not a member of X, it is not in the scope of X, and it cannot be called with the x.F and xptr->F selector operators (where x is an X object and xptr is a pointer to an X object).

If the specifier **friend** is used with a function declaration or definition within the class X, it becomes a friend of X.

friend functions defined within a class obey the same inline rules as member functions (see Inline functions). **friend** functions are not affected by their position within the class or by any access specifiers. For example:

```
class X {
   int i;                           // private to X
   friend void friend_func(X*, int);
/* friend_func is not private, even though it's declared in the private section */
public:
     void member_func(int);
};
/* definitions; note both functions access private int i */
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }
X xobj;
/* note difference in function calls */
friend_func(&xobj, 6);
xobj.member_func(6);
```

You can make all the functions of class Y into friends of class X with a single declaration:

```
class Y;                        // incomplete declaration
class X {
   friend Y;
   int i;
   void member_funcX();
};
class Y; {                      // complete the declaration
   void friend_X1(X&);
   void friend_X2(X*);
.
.
.
};
```

The functions declared in Y are friends of X, although they have no **friend** specifiers. They can access the private members of X, such as i and member_funcX.

It is also possible for an individual member function of class X to be a friend of class Y:

```
class X {
.
.
.
   void member_funcX();
```

```
}
class Y {
   int i;
   friend void X::member_funcX();
.
.
.
};
```

Class friendship is not transitive: X friend of Y and Y friend of Z does not imply X friend of Z. Friendship is not inherited.

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes

3.1.3.1.1.3 Class Member List

The optional member-list is a sequence including, but not exclusive to:

- Data declarations (of any type, including enumerations, bit fields and other classes)
- Nested type declarations
- Nested type definitions
- Template declarations
- Template definitions
- Function declarations
- Function definitions
- Constructors
- A destructor

Members of a class can optional have storage class specifiers and access modifiers. The objects thus defined are called class members. The storage class specifiers **auto**, **extern**, and **register** are not allowed. Members can be declared with the **static** storage class specifiers.

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.4 Inline Functions

This section contains Inline Function topics.

Topics

Name	Description
Inline Functions (see page 391)	You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an inline function. The compiler can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an inline expansion of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The **inline** specifier indicates to the compiler you would like an... more (see page 391)

3.1.3.1.1.4.1 Inline Functions

You can declare a member function within its class and define it elsewhere. Alternatively, you can both declare and define a member function within its class, in which case it is called an inline function.

The compiler can sometimes reduce the normal function call overhead by substituting the function call directly with the compiled code of the function body. This process, called an inline expansion of the function body, does not affect the scope of the function name or its arguments. Inline expansion is not always possible or feasible. The **inline** specifier indicates to the compiler you would like an inline expansion.

Note: The compiler can ignore requests for inline expansion.

Explicit and implicit **inline** requests are best reserved for small, frequently used functions, such as the operator functions that implement overloaded operators. For example, the following class declaration of func:

```
int i;                          // global int
class X {
public:
```

```
   char* func(void) { return i; }  // inline by default
   char* i;
};
```

is equivalent to:

```
inline char* X::func(void) { return i; }
```

func is defined outside the class with an explicit **inline** specifier. The value i returned by func is the **char*** i of class X (see Member scope).

Inline functions and exceptions

An inline function with an exception-specification will never be expanded inline by the compiler. For example,

```
inline void f1() throw(int)
{
   // Warning: Functions with exception specifications are not expanded inline
}
```

The remaining restrictions apply only when destructor cleanup is enabled.

Note: Destructors are called by default. See Setting exception handling options for information about exception-handling switches.

An inline function that takes at least one parameter that is of type 'class with a destructor' will not be expanded inline. Note that this restriction does not apply to classes that are passed by reference. Example:

```
struct foo {
   foo();
   ~foo();
};
inline void f2(foo& x)
{
   // no warning, f2() can be expanded inline
}
inline void f3(foo x)
{
   // Warning: Functions taking class-by-value argument(s) are
   //          not expanded inline in function f3(foo)
}
```

An inline function that returns a class with a destructor by value will not be expanded inline whenever there are variables or temporaries that need to be destructed within the return expression:

```
struct foo {
   foo();
   ~foo();
};
inline foo f4()
{
   return foo();
   // no warning, f4() can be expanded inline
}
inline foo f5()
{
   foo X;
   return foo(); // Object X needs to be destructed
   // Warning: Functions containing some return statements are
   //          not expanded inline in function f5()
}
inline foo f6()
{
   return ( foo(), foo() );  // temporary in return value
   // Warning:Functions containing some return statements are
   //         not expanded inline in function f6()
}
```

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.5 Class Name Scope

The scope of a class name is local. There are some special requirements if the class name appears more than once in the same scope. Class name scope starts at the point of declaration and ends with the enclosing block. A class name hides any class, object, enumerator, or function with the same name in the enclosing scope. If a class name is declared in a scope containing the declaration of an object, function, or enumerator of the same name, the class can be referred to only by using the elaborated type specifier. This means that the class key, **class**, **struct**, or **union**, must be used with the class name. For example,

```
struct S { ... };
int S(struct S *Sptr);
void func(void) {
   S t;         // ILLEGAL declaration: no class key and function S in scope
   struct S s;  // OK: elaborated with class key
   S(&s);       // OK: this is a function call
}
```

C++ also allows a forward class declaration:

```
class X;  // no members, yet!
```

Forward declarations permit certain references to class name X (usually references to pointers to class objects) before the class has been fully defined. See Structure member declarations for more information. Of course, you must make a complete class declaration with members before you can define and use objects of that class.

See also the syntax for forward declarations of VCL classes.

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.6 The Keyword This

This section contains Keyword this topics.

Topics

Name	Description
Static Members (see page 394)	The storage class specifier **static** can be used in class declarations of data and function members. Such members are called static members and have distinct properties from nonstatic members. With nonstatic members, a distinct copy “exists” for each instance of the class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If x is a static member of class X, it can be referenced as X::x (even if objects of class X haven’t been created yet). It is still possible to access x using the normal member access... more (see page 394)

3.1.3.1.1.6.1 Static Members

The storage class specifier **static** can be used in class declarations of data and function members. Such members are called static members and have distinct properties from nonstatic members. With nonstatic members, a distinct copy “exists” for each instance of the class; with static members, only one copy exists, and it can be accessed without reference to any particular object in its class. If x is a static member of class X, it can be referenced as X::x (even if objects of class X haven’t been created yet). It is still possible to access x using the normal member access operators. For example, y.x and yptr->x, where y is an object of class X and yptr is a pointer to an object of class X, although the expressions y and yptr are not evaluated. In particular, a static member function can be called with or without the special member function syntax:

```
class X {
   int member_int;
public:
   static void func(int i, X* ptr);
};
void g(void)
{
   X obj;
   func(1, &obj);       // error unless there is a global func()
                        // defined elsewhere
   X::func(1, &obj);    // calls the static func() in X
                        // OK for static functions only
   obj.func(1, &obj);   // so does this (OK for static and
                        // nonstatic functions)
}
```

Because static member functions can be called with no particular object in mind, they don’t have a **this** pointer, and therefore

cannot access nonstatic members without explicitly specifying an object with **.** or **->**. For example, with the declarations of the previous example, func might be defined as follows:

```
void X::func(int i, X* ptr)
{
   member_int = i;       // which object does member_int
                         // refer to? Error
   ptr->member_int = i;  // OK: now we know!
}
```

Apart from inline functions, static member functions of global classes have external linkage. Static member functions cannot be virtual functions. It is illegal to have a static and nonstatic member function with the same name and argument types.

The declaration of a static data member in its class declaration is not a definition, so a definition must be provided elsewhere to allocate storage and provide initialization.

Static members of a class declared local to some function have no linkage and cannot be initialized. Static members of a global class can be initialized like ordinary global objects, but only in file scope. Static members, nested to any level, obey the usual class member access rules, except they can be initialized.

```
class X {
   static int x;
   static const int size =  5;
   class inner {
      static float f;
      void func(void);     // nested declaration
      };
public :
   char array[size];
};
int X::x = 1;
float X::inner::f = 3.14;  // initialization of nested static
void X::inner::func(void) {     /*  define the nested function */  }
```

The principal use for static members is to keep track of data common to all objects of a class, such as the number of objects created, or the last-used resource from a pool shared by all such objects. Static members are also used to

- Reduce the number of visible global names
- Make obvious which static objects logically belong to which class
- Permit access control to their names

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.7 Class Names

class-name is any identifier unique within its scope. With structures, classes, and unions, class-name can be omitted. See Untagged structures and typedefs for discussion of untagged structures.

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.8 Member Scope

This section contains Member Scope topics.

Topics

Name	Description
Base And Derived Class Access (see page 397)	When you declare a derived class D, you list the base classes B1, B2, ... in a comma-delimited base-list:
Member Access Control (see page 399)	Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: **public**, **private**, and **protected**. The significance of these attributes is as follows: • **public**: The member can be used by any function. • **private**: The member can be used only by member functions and friends of the class it's declared in. • **protected**: Same as for **private**. Additionally, the member can be used by member functions and friends of classes derived from the declared class, but only in... more (see page 399)

Member Scope (see page 400)	The expression X::func() in the example in Inline functions and exceptions uses the class name X with the scope access modifier to signify that func, although defined “outside” the class, is indeed a member function of X and exists within the scope of X. The influence of X:: extends into the body of the definition. This explains why the i returned by func refers to X::i, the **char*** i of X, rather than the global **int** i. Without the X:: modifier, the function func would represent an ordinary non-class function, returning the global **int** i. All member functions, then,... more (see page 400)
Nested Types (see page 401)	Tag or **typedef** names declared inside a class lexically belong to the scope of that class. Such names can, in general, be accessed only by using the **xxx::yyy** notation, except when in the scope of the appropriate class. A class declared within another class is called a nested class. Its name is local to the enclosing class; the nested class is in the scope of the enclosing class. This is a purely lexical nesting. The nested class has no additional privileges in accessing members of the enclosing class (and vice versa). Classes can be nested in this way to... more (see page 401)

3.1.3.1.1.8.1 Base And Derived Class Access

When you declare a derived class D, you list the base classes B1, B2, ... in a comma-delimited base-list:

```
class-key D : base-list { <member-list> }
```

D inherits all the members of these base classes. (Redefined base class members are inherited and can be accessed using scope overrides, if needed.) D can use only the **public** and **protected** members of its base classes. But, what will be the access attributes of the inherited members as viewed by D? D might want to use a **public** member from a base class, but make it **private** as far as outside functions are concerned. The solution is to use access specifiers in the base-list.

Note: Since a base class can itself be a derived class, the access attribute question is recursive: you backtrack until you reach the basemost of the base classes, those that do not inherit.

When declaring D, you can use the access specifier **public**, **protected**, or **private** in front of the classes in the base-list:

```
class D : public B1, private B2, ... {
   .
   .
   .
}
```

These modifiers do not alter the access attributes of base members as viewed by the base class, though they can alter the access attributes of base members as viewed by the derived class.

The default is private if D is a class declaration, and public if D is a struct declaration.

Note: Unions cannot have base classes, and unions cannot be used as base classes.

The derived class inherits access attributes from a base class as follows:

- **public** base class: **public** members of the base class are **public** members of the derived class. **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **protected** base class: Both **public** and **protected** members of the base class are **protected** members of the derived class. **private** members of the base class remain **private** to the base class.
- **private** base class: Both **public** and **protected** members of the base class are **private** members of the derived class. **private** members of the base class remain **private** to the base class.

Note that **private** members of a base class are always inaccessible to member functions of the derived class unless **friend** declarations are explicitly declared in the base class granting access. For example,

```
/* class X is derived from class A */
class X : A {               // default for class is private A
   .
   .
   .
}
/* class Y is derived (multiple inheritance) from B and C
```

```
   B defaults to private B */
class Y : B, public C {     // override default for C
  .
  .
  .
}
/* struct S is derived from D */
struct S : D {              // default for struct is public D
  .
  .
  .
}
/* struct T is derived (multiple inheritance) from D and E
   E defaults to public E */
struct T : private D, E {   // override default for D
                            // E is public by default
  .
  .
  .
}
```

The effect of access specifiers in the base list can be adjusted by using a qualified-name in the public or protected declarations of the derived class. For example:

```
class B {
   int a;               // private by default
public:
   int b, c;
   int Bfunc(void);
};
class X : private B {   // a, b, c, Bfunc are now private in X
   int d;               // private by default, NOTE: a is not
                        // accessible in X
public:
   B::c;                // c was private, now is public
   int e;
   int Xfunc(void);
};
int Efunc(X& x);        // external to B and X
```

The function Efunc() can use only the public names c, e, and Xfunc().

The function Xfunc() is in X, which is derived from **private** B, so it has access to

- The “adjusted-to-public” c
- The “private-to-X” members from B: b and Bfunc()
- X’s own private and public members: d, e, and Xfunc()

However, Xfunc() cannot access the “private-to-B” member, a.

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.8.2 Member Access Control

Members of a class acquire access attributes either by default (depending on class key and declaration placement) or by the use of one of the three access specifiers: **public**, **private**, and **protected**. The significance of these attributes is as follows:

- **public**: The member can be used by any function.
- **private**: The member can be used only by member functions and friends of the class it's declared in.
- **protected**: Same as for **private**. Additionally, the member can be used by member functions and friends of classes derived from the declared class, but only in objects of the derived type. (Derived classes are explained in Base and derived class access.)

  Note: Friend function declarations are not affected by access specifiers (see Friends of classes for more information).

  Members of a class are **private** by default, so you need explicit **public** or **protected** access specifiers to override the default.

  Members of a **struct** are **public** by default, but you can override this with the **private** or **protected** access specifier.

  Members of a **union** are **public** by default; this cannot be changed. All three access specifiers are illegal with union members.

  A default or overriding access modifier remains effective for all subsequent member declarations until a different access modifier is encountered. For example,

```
class X {
   int i;    // X::i is private by default
   char ch;  // so is X::ch
public:
   int j;    // next two are public
   int k;
protected:
   int l;    // X::l is protected
};
struct Y {
   int i;    // Y::i is public by default
private:
   int j;    // Y::j is private
public:
   int k;    // Y::k is public
};
union Z {
   int i;    // public by default; no other choice
   double d;
};
```

Note: The access specifiers can be listed and grouped in any convenient sequence. You can save typing effort by declaring all the private members together, and so on.

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.8.3 Member Scope

The expression X::func() in the example in Inline functions and exceptions uses the class name X with the scope access modifier to signify that func, although defined “outside” the class, is indeed a member function of X and exists within the scope of X. The influence of X:: extends into the body of the definition. This explains why the i returned by func refers to X::i, the **char*** i of X, rather than the global **int** i. Without the X:: modifier, the function func would represent an ordinary non-class function, returning the global **int** i.

All member functions, then, are in the scope of their class, even if defined outside the class.

Data members of class X can be referenced using the selection operators **.** and **->** (as with C structures). Member functions can also be called using the selection operators (see The keyword this). For example:

```
class X {
public:
   int i;
   char name[20];
   X *ptr1;
   X *ptr2;
   void Xfunc(char*data, X* left, X* right);   // define elsewhere
};
void f(void);
{
   X x1, x2, *xptr=&x1;
   x1.i = 0;
   x2.i = x1.i;
   xptr->i = 1;
   x1.Xfunc("stan", &x2, xptr);
}
```

If m is a member or base member of class X, the expression X::m is called a qualified name; it has the same type as m, and it is an lvalue only if m is an lvalue. It is important to note that, even if the class name X is hidden by a non-type name, the qualified name X::m will access the correct class member, m.

Class members cannot be added to a class by another section of your program. The class X cannot contain objects of class X, but can contain pointers or references to objects of class X (note the similarity with C’s structure and union types).

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.8.4 Nested Types

Tag or **typedef** names declared inside a class lexically belong to the scope of that class. Such names can, in general, be accessed only by using the **xxx::yyy** notation, except when in the scope of the appropriate class.

A class declared within another class is called a nested class. Its name is local to the enclosing class; the nested class is in the scope of the enclosing class. This is a purely lexical nesting. The nested class has no additional privileges in accessing members of the enclosing class (and vice versa).

Classes can be nested in this way to an arbitrary level, up to the limits of memory. Nested classes can be declared inside some class and defined later. For example,

```
struct outer
{
   typedef int t;  // 'outer::t' is a typedef name
   struct inner    // 'outer::inner' is a class
   {
      static int x;
   };
   static int x;
      int f();
   class deep;     // nested declaration
};
int outer::x;     // define static data member
int outer::f() {
   t x;           // 't' visible directly here
   return x;
   }
int outer::inner::x;    // define static data member
outer::t x;             //  have to use 'outer::t' here
class outer::deep { };  // define the nested class here
```

With CodeGear C++ 2.0, any tags or **typedef** names declared inside a class actually belong to the global (file) scope. For

example:

```
struct foo
{
   enum bar { x };   // 2.0 rules: 'bar' belongs to file scope
                     // 2.1 rules: 'bar' belongs to 'foo' scope
};
bar x;
```

The preceding fragment compiles without errors. But because the code is illegal under the 2.1 rules, a warning is issued as follows:

```
Warning: Use qualified name to access nested type 'foo::bar'
```

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.9 Class Objects

Class objects can be assigned (unless copying has been restricted), passed as arguments to functions, returned by functions (with some exceptions), and so on. Other operations on class objects and members can be user-defined in many ways, including definition of member and friend functions and the redefinition of standard functions and operators when used with objects of a certain class.

Redefined functions and operators are said to be overloaded. Operators and functions that are restricted to objects of a certain class (or related group of classes) are called member functions for that class. C++ offers the overloading mechanism that allows the same function or operator name can be called to perform different tasks, depending on the type or number of arguments or operands.

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.10 Virtual Base Classes

This section contains Virtual Base Class topics.

Topics

Name	Description
Virtual Base Classes (see page 403)	A **virtual** class is a base class that is passed to more than one derived class, as might happen with multiple inheritance. You cannot specify a base class more than once in a derived class:

3.1.3.1.1.10.1 Virtual Base Classes

A **virtual** class is a base class that is passed to more than one derived class, as might happen with multiple inheritance.

You cannot specify a base class more than once in a derived class:

```
class B { ... };
class D : B; B { ... }; //ILLEGAL
```

However, you can indirectly pass a base class to the derived class more than once:

```
class X : public B { ... };
class Y : public B { ... };
class Z : public X, public Y { ... }; //OK
```

In this case, each object of class Z has two sub-objects of class B.

If this causes problems, add the keyword **virtual** to the base class specifier. For example,

```
class X : virtual public B { ... };
class Y : virtual public B { ... };
class Z : public X, public Y { ... };
```

B is now a virtual base class, and class Z has only one sub-object of class B.

Constructors for Virtual Base Classes

Constructors for virtual base classes are invoked before any non-virtual base classes.

If the hierarchy contains multiple virtual base classes, the virtual base class constructors invoke in the order they were declared.

Any non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first, so that the virtual base class can be properly

constructed. For example, this code

```
class X : public Y , virtual public Z
 X one;
```

produces this order:

```
Z(); // virtual base class initialization
Y(); // non-virtual base class
X(); // derived class
```

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes

Friends Of Classes (see page 389)

3.1.3.1.1.11 Class Types

The declaration creates a unique type, class type class-name. This lets you declare further class objects (or instances) of this type, and objects derived from this type (such as pointers to, references to, arrays of class-name, and so on):

```
class X { ... };
X x, &xr, *xptr, xarray[10];
/* four objects: type X, reference to X, pointer to X and array of X */
struct Y { ... };
Y y, &yr, *yptr, yarray[10];
// C would have
// struct Y y, *yptr, yarray[10];
union Z { ... };
Z z, &zr, *zptr, zarray[10];
// C would have
// union Z z, *zptr, zarray[10];
```

Note the difference between C and C++ structure and union declarations: The keywords **struct** and **union** are essential in C, but in C++, they are needed only when the class names, Y and Z, are hidden (see Class name scope)

See Also

C++ Classes (see page 387)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.12 Member Functions

A function declared without the **friend** specifier is known as a member function of the class. Functions declared with the **friend** modifier are called friend functions.

Member functions are often referred to as methods in Delphi documentation.

The same name can be used to denote more than one function, provided they differ in argument type or number of arguments.

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Objects (see page 402)

Class Member List (see page 390)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.13 VCL Class Declarations

Syntax

```
__declspec(<decl-modifier>)
```

Description

The decl-modifier argument can be **delphiclass** or **pascalimplementation**. These arguments should be used only with classes derived from VCL classes.

- You must use __declspec(delphiclass) for any forward declaration of classes that are directly or indirectly derived from TObject.
- Use the __declspec(pascalimplementation) modifier to indicate that a class has been implemented in the Delphi language. This modifier appears in a Delphi portability header file with a .hpp extension.

  Note: Another argument, delphireturn

  , is used internally to mark C++ classes for VCL-compatible handling in function calls as parameters and return values. The **delphiclass** argument is used to create classes that have the following VCL compatibility.
- VCL-compatible RTTI
- VCL-compatible constructor/destructor behavior
- VCL-compatible exception handling

A VCL-compatible class has the following restrictions.

- No virtual base classes or multiple inheritance is allowed.
- Must be dynamically allocated by using the global **new** operator.
- Copy and assignment constructors must be explicitly defined. The compiler does not automatically provide these constructors for VCL-derived classes.
- Must publicly inherit from another VCL class.

3.1.3.1.1.14 Class Methods

In C++Builder, a **class method** is a method that can be invoked on a class name, as well as on an instance of that class. In contrast, object methods can be invoked only on objects -- instances of a class.

How Class Methods Work in Delphi

The Delphi language (Object Pascal) supports class methods and a metaclass as described in the Class Methods section in the Delphi help topic Methods.

Here is a sample:

```
TTest = class;

// Declaring metaclass type
TTestClass = class of TTest;

TTest = class
public
  // Class methods
  class function add(I, J: Integer): Integer;
  class function GetClsName: string;

  // Virtual class method
  class function GetClsNameVirt: string; virtual;

  // Class getter and setter
  class function GetCount: Integer;
  class procedure SetCount(I: Integer);

  // Virtual class getter and setter
  class function GetStrProp: string; virtual;
  class procedure SetStrProp(N: string); virtual;
```

```
    // Class static
    class function GetStaticCount: Integer; static;
    class procedure SetStaticCount(I: Integer); static;

    // Class properties
    property Count: Integer read GetCount write SetCount; // Non-virtual
    property StrProp: string read GetStrProp write SetStrProp; // Virtual g/setters
  end;

  // Function that takes a class reference
  function RegisterTestType(Cls: TTestClass): boolean;
```

Class Methods as Static Methods with Explicit TMetaClass* Parameter

Prior to C++Builder 2009, class methods were represented as static methods with an explicit metaclass parameter. A metaclass is represented by a pointer to a TMetaClass instance or a TClass. This metaclass or class reference is obtained with the __classid extension, which returns a TMetaClass* instance for a class name.

Here is an example of this usage, attempting to define the same methods and properties as in the Delphi sample above. Note that some of the Delphi class method features cannot be done properly using this metaclass approach.

```
// All metaclass types are TMetaClass* in C++
typedef TMetaClass* TTestClass;

class DELPHICLASS TTest;
class PASCALIMPLEMENTATION TTest : public System::TObject
{
  typedef System::TObject inherited;
  public:
      // Class methods exposed as static methods with the 'hidden'
      // class reference explicit as the first parameter.
      static int __fastcall add(TMetaClass* vmt, int I, int J);
      static UnicodeString __fastcall GetClsName(TMetaClass* vmt);

      // Virtual class methods would be exposed as plain virtual methods with
      // the hidden class reference explicit as the first parameter.
      // This means that upon calling this method from C++, there would have
      // to be two 'hidden' parameters passed in--which would not work.
      virtual UnicodeString __fastcall GetClsNameVirt(TMetaClass* vmt);

      // Non-virtual methods are feasible to work with. These two methods
      // are typically overloaded with the first TMetaClass* parameter
      // hardcoded to __classid(TTest).
      static int __fastcall GetCount(TMetaClass* vmt);
      static void __fastcall SetCount(TMetaClass* vmt, int I);

      // You can overload these virtual setters and getters, but given
      // that the call is incorrect, the program fails
      // when accessing the property tied to these methods.
      virtual UnicodeString __fastcall GetStrProp(TMetaClass* vmt);
      virtual void __fastcall SetStrProp(TMetaClass* vmt, UnicodeString N);

      // Delphi class static method would be plain C++ static.
      static int __fastcall GetStaticCount();
      static void __fastcall SetStaticCount(int I);

      // Although the compiler allows these declarations,
      // because TMetaClass* is required, you'll get an error
      // from the C++ compiler upon attempting to access these
properties.
      __property int Count = {read=GetCount, write=SetCount, nodefault};
      __property UnicodeString StrProp = {read=GetStrProp, write=SetStrProp};};

extern PACKAGE bool __fastcall RegisterTestType(TMetaClass* Cls);
```

There are several limitations with this way of exposing class methods using an explicit TMetaClass* parameter:

- C++ code cannot properly invoke virtual class methods. C++ calls would have to pass two hidden this parameters: the pointer to the instance of the object and the explicit `TMetaClass*` parameter. However, the function expects only one `TMetaClass*` parameter.
- Even in cases where the call to a virtual class method succeeds, the C++ code must have an instance of the object to invoke the method, but a proper class method should be invokable without requiring an object instance.
- C++ code cannot properly access properties whose getters or setters are class methods, because there's no way to provide the `TMetaClass*` parameter, which must be explicit.

Class Methods Using __classmethod Keyword

C++Builder 2009 introduces class methods that remove the limitations listed above and provide a simpler and more intuitive syntax for class methods. Class methods are now declared with the new keyword **__classmethod**.

Here is how the code above is written using the **__classmethod** syntax:

```
// Class references still use TMetaClass*.
typedef TMetaClass* TTestClass;

class DELPHICLASS TTest;
class PASCALIMPLEMENTATION TTest : public System::TObject
{
  typedef System::TObject inherited;

  public:
      // The TMetaClass* parameter is now hidden.
      // The __classmethod keyword flags methods as class methods.
      __classmethod int __fastcall add(int I, int J);
      __classmethod UnicodeString __fastcall GetClsName();

      // Virtual methods can be used.
      __classmethod virtual UnicodeString __fastcall GetClsNameVirt();

      // Methods can access class properties
      __classmethod int __fastcall GetCount();
      __classmethod void __fastcall SetCount(int I);
      __classmethod virtual UnicodeString __fastcall GetStrProp();
      __classmethod virtual void __fastcall SetStrProp(UnicodeString N);

      // Class static methods still map to C++ static methods.
      static int __fastcall GetstaticCount();
      static void __fastcall SetstaticCount(int I);

      // Class properties
      __property int Count = {read=GetCount, write=SetCount, nodefault};
      __property UnicodeString StrProp = {read=GetStrProp, write=SetStrProp};
};
```

C++Builder does not provide for a way to declare a class reference other than by `TMetaClass*`. Hence, class methods are invoked with a typename or with an instance of a object. Virtual class methods are invoked the same way regular virtual methods are invoked, except that instead of the this pointer being pushed, the metaclass pointer is the hidden parameter.

This implementation provides two capabilities:

- Class methods can be virtual.
- Class properties can be defined.

Dynamic Dispatch of Virtual Class Methods

If you have a class method defined with **__classmethod** for a class that is derived from another class, you can't dynamically invoke the derived class's virtual method using a class name. However, you can invoke the proper method using an instance.

The "virtual" mechanism of class methods is analogous to virtual for regular methods. For regular methods, you get polymorphism only when using a pointer or a reference, because the vtable is then determined at run time. With a value instance, you don't get polymorphism because the vtable is determine at compile time.

Similarly, for class methods you get polymorphism with an instance but not with a type. To illustrate:

```
class TBase {
    virtual __classmethod void cfunc();
    virtual  void vfunc();
};

class TDerived: TBase {
    virtual __classmethod void cfunc();
    virtual void vfunc();
};

// Regular virtual methods
TDerived d;
d.vfunc();   //calls TDerived::vfunc;

TBase* bp = new TDerived();
bp->vfunc(); //calls TDerived::vfunc;

TBase& br = TDerived();
br.vfunc();  //calls TDerived::vfunc;

TBase b;
b.vfunc();   //calls TBase::vfunc
// Class virtual methods
TBase* b = new TDerived();
b->cfunc();         //calls version in TDerived--dynamic
TDerived::cfunc();  //calls version in TDerived--compile time--not dynamic
__classid(TDerived)->cfunc();  //compiler error
```

Updating Your Code to Use __classmethod

If you have C++ code that implements class methods using the older style class methods with a Metaclass * parameter, you can update your code to use the __classmethod syntax. Since the TMetaClass* parameter is now hidden, the signature and mangling of such class methods are different. The following table illustrates how to update common C++ constructs using class methods. It uses the add(int i, int j) method and the Count property from the __classmethod code sample above to illustrate the changes you would make.

Purpose of code	Old style “class static” code	Updated code using __classmethod
Function declaration	class TTest : public System::TObject { public: static int __fastcall add(TMetaClass* vmt, int I, int J); };	class TTest : public System::TObject { public: __classmethod int __fastcall add(int I, int J); };
Using __classid	TTest::add(__classid(TTest), 10, 20);	TTest::add(10, 20);
Using derived __classid	TTestDerived::add(__classid(TTestDerived), 10, 20);	TTestDerived::add(10, 20);
Using class instance	TTest* p = new TTest(); ... TTest::add(p->ClassType(), 10, 20);	TTest* p = new TTest(); ... p->add(10, 20);
Using derived instance	TTest* p = new TTestDerived(); ... TTest::add(p->ClassType(), 10, 20);	TTest* p = new TTestDerived(); ... p->add(10, 20);
Using class property	TTest* p = new TTest(); ... p->Count = 20;	TTest::Count = 20; //Using class name // Using instance TTest* p = new TTest(); ... p->Count = 20;

See Also

C++ Classes (see page 387)

Class Types (see page 404)

Class Scope (see page 431)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.1.15 Static Properties

Static properties have been implemented in C++Builder 2009 to enhance compatibility with the Delphi language

Declaring Class Functions as Static

If property access functions (getters and setters) are declared as static, you can use the property directly from the class without specifying an object. For example:

```
class TMyClass : TComponent {
  static int x;
  static int GetX() { return x; }
  static void SetX( int i ) { x = i; }

  __property int X = { read = GetX, write = SetX }
};

TMyClass::X = 5;
int y = TMyClass::X;
```

Delphi also allows plain static methods as getters and setters of properties. Note that these static methods are not class methods; there is no hidden parameter. For example:

```
class TTest{
  static int __fastcall GetCount();
  static void __fastcall SetCount(int I);
public:
  __property int Count = {read=GetCount, write=SetCount};
};
```

In this case, you can then use the properties as this:

```
int c = TTest::Count;
TTest t;
int i = t.Count;
```

The C++ compiler already supports static properties that map to a member variable. For example:

```
class TEncoding  {
   static int Prop;
 public:
   __property int Count = { read=Prop };
};
int f(TEncoding& t) {
   return t.Count;
}
```

See Also

C++ Classes (see page 387)

Class Methods (see page 406)

Class Types (see page 404)

Class Scope (see page 431)

Class Member List (see page 390)

Member Functions (see page 405)

The Keyword This (see page 394)

Static Members (see page 394)

Inline Functions (see page 391)

Member Scope (see page 400)

Nested Types (see page 401)

Member Access Control (see page 399)

Base And Derived Class Access (see page 397)

Virtual Base Classes (see page 403)

Friends Of Classes (see page 389)

3.1.3.1.2 Class Templates Overview

This section contains Class Templates Overview topics.

Topics

Name	Description
Using Angle Brackets In Templates (see page 412)	Be careful when using the right angle bracket character upon instantiation:
Template Arguments (see page 412)	Multiple arguments are allowed as part of the class template declaration. Template arguments can also represent values in addition to data types:
Class Templates (see page 412)	A class template (also called a generic class or class generator) lets you define a pattern for class definitions. Consider the following example of a vector class (a one-dimensional array). Whether you have a vector of integers or any other type, the basic operations performed on the type are the same (insert, delete, index, and so on). With the element type treated as a type parameter to the class, the system will generate type-safe class definitions on the fly. As with function templates, an explicit template class specialization can be provided to override the automatic definition for a given type:... more (see page 412)
Eliminating Pointers In Templates (see page 412)	Another design technique is to include actual objects, making pointers unnecessary. This can also reduce the number of **virtual** function calls required, since the compiler knows the actual types of the objects. This is beneficial if the **virtual** functions are small enough to be effectively inlined. It's difficult to inline **virtual** functions when called through pointers, because the compiler doesn't know the actual types of the objects being pointed to.

Using Type-safe Generic Lists In Templates (see page 413)	In general, when you need to write lots of nearly identical things, consider using templates. The problems with the following class definition, a generic list class,

3.1.3.1.2.1 Using Angle Brackets In Templates

Be careful when using the right angle bracket character upon instantiation:

```
Buffer<char, (x > 100 ? 1024 : 64)> buf;
```

In the preceding example, without the parentheses around the second argument, the > between x and 100 would prematurely close the template argument list.

3.1.3.1.2.2 Template Arguments

Multiple arguments are allowed as part of the class template declaration. Template arguments can also represent values in addition to data types:

```
template<class T, int size = 64> class Buffer { ... };
```

Both non-type template arguments such as size and type arguments can have default values. The value supplied for a non-type template argument must be a constant expression:

```
const int N = 128;
int i = 256;
Buffer<int, 2*N> b1;// OK
Buffer<float, i> b2;// Error: i is not constant
```

Since each instantiation of a template class is indeed a class, it receives its own copy of static members. Similarly, template functions get their own copy of static local variables.

3.1.3.1.2.3 Class Templates

A class template (also called a generic class or class generator) lets you define a pattern for class definitions. Consider the following example of a vector class (a one-dimensional array). Whether you have a vector of integers or any other type, the basic operations performed on the type are the same (insert, delete, index, and so on). With the element type treated as a type parameter to the class, the system will generate type-safe class definitions on the fly.

As with function templates, an explicit template class specialization can be provided to override the automatic definition for a given type:

```
class Vector<char *> { ... };
```

The symbol Vector must always be accompanied by a data type in angle brackets. It cannot appear alone, except in some cases in the original template definition.

See Also

Exporting And Importing Templates (see page 432)

Eliminating Pointers In Templates (see page 412)

Template Arguments (see page 412)

3.1.3.1.2.4 Eliminating Pointers In Templates

Another design technique is to include actual objects, making pointers unnecessary. This can also reduce the number of **virtual** function calls required, since the compiler knows the actual types of the objects. This is beneficial if the **virtual** functions are small enough to be effectively inlined. It's difficult to inline **virtual** functions when called through pointers, because the compiler doesn't know the actual types of the objects being pointed to.

```
template <class T> aBase {
```

```
    .
    .
    .
 private:
  T buffer;
};
class anObject : public aSubject, public aBase<aFilebuf> {
    .
    .
    .
};
```

All the functions in aBase can call functions defined in aFilebuf directly, without having to go through a pointer. And if any of the functions in aFilebuf can be inlined, you'll get a speed improvement, because templates allow them to be inlined.

3.1.3.1.2.5 Using Type-safe Generic Lists In Templates

In general, when you need to write lots of nearly identical things, consider using templates. The problems with the following class definition, a generic list class,

```
class GList
{
 public:
   void insert( void * );
   void *peek();
    .
    .
    .
};
```

are that it isn't type-safe and common solutions need repeated class definitions. Since there's no type checking on what gets inserted, you have no way of knowing what results you'll get. You can solve the type-safe problem by writing a wrapper class:

```
class FooList : public GList {
 public:
   void insert( Foo *f ) { GList::insert( f ); }
   Foo *peek() { return (Foo *)GList::peek(); }
      .
      .
      .
};
```

This is type-safe. insert will only take arguments of type pointer-to-Foo or object-derived-from-Foo, so the underlying container will only hold pointers that in fact point to something of type Foo. This means that the cast in FooList::peek() is always safe, and you've created a true FooList. Now, to do the same thing for a BarList, a BazList, and so on, you need repeated separate class definitions. To solve the problem of repeated class definitions and type-safety, you can once again use templates. See the example for type-safe generic list class. The C++ Standard Template Library (STL) has a rich set of type-safe collection classes.

By using templates, you can create whatever type-safe lists you want, as needed, with a simple declaration. And there's no code generated by the type conversions from each wrapper class so there's no runtime overhead imposed by this type safety.

3.1.3.1.3 Compiler Template Switches

This section contains Compiler Template Switch topics.

Topics

Name	Description
Template Compiler Switches (see page 414)	The -Jg family of switches control how instances of templates are generated by the compiler. Every template instance encountered by the compiler will be affected by the value of the switch at the point where the first occurrence of that particular instance is seen by the compiler. For template functions the switch applies to the function instances; for template classes, it applies to all member functions and static data members of the template class. In all cases, this switch applies only to compiler-generated template instances and never to user-defined instances. It can be used, however, to tell the compiler which... more (see page 414)
Template Generation Semantics (see page 414)	The C++ compiler generates the following methods for template instances: • Those methods which were actually used • Virtual methods of an instance • All methods of explicitly instantiated classes The advantage of this behavior is that it results in significantly smaller object files, libraries and executable files, depending on how heavily you use templates. Optionally, you can use the '-Ja' switch to generate all methods. You can also force all of the out-of-line methods of a template instance to be generated by using the explicit template instantiation syntax defined in the ISO/ANSI C++ Standard. The syntax is: **template class** classname**<**template... more (see page 414)

3.1.3.1.3.1 Template Compiler Switches

The -Jg family of switches control how instances of templates are generated by the compiler. Every template instance encountered by the compiler will be affected by the value of the switch at the point where the first occurrence of that particular instance is seen by the compiler.

For template functions the switch applies to the function instances; for template classes, it applies to all member functions and static data members of the template class. In all cases, this switch applies only to compiler-generated template instances and never to user-defined instances. It can be used, however, to tell the compiler which instances will be user-defined so that they aren't generated from the template.

3.1.3.1.3.2 Template Generation Semantics

The C++ compiler generates the following methods for template instances:

- Those methods which were actually used
- Virtual methods of an instance
- All methods of explicitly instantiated classes

The advantage of this behavior is that it results in significantly smaller object files, libraries and executable files, depending on how heavily you use templates.

Optionally, you can use the '-Ja' switch to generate all methods.

You can also force all of the out-of-line methods of a template instance to be generated by using the explicit template instantiation syntax defined in the ISO/ANSI C++ Standard. The syntax is:

template class classname**<**template parameter**>;**

The following STL example directs the compiler to generate all out-of-line methods for the “list<char>” class, regardless of whether they are referenced by the user’s code:

```
template class list<char>
```

You can also explicitly instantiate a single method, or a single static data member of a template class, which means that the method is generated to the .OBJ even though it is not used:

```
template void classname <template parameter>:: methodname ();
```

3.1.3.1.4 Constructors And Destructors

This section contains Constructor and Destructor topics.

Topics

Name	Description
Introduction To Constructors And Destructors (see page 415)	There are several special member functions that determine how the objects of a class are created, initialized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features: • They do not have return value declarations (not even **void**). • They cannot be inherited, though a derived class can call the base class’s constructors and destructors. • Constructors, like most C++ functions, can have default arguments or use member... more (see page 415)
Constructors (see page 416)	This section contains Constructor topics.
Destructors (see page 422)	This section contains Destructor topics.

3.1.3.1.4.1 Introduction To Constructors And Destructors

There are several special member functions that determine how the objects of a class are created, initialized, copied, and destroyed. Constructors and destructors are the most important of these. They have many of the characteristics of normal member functions—you declare and define them within the class, or declare them within the class and define them outside—but they have some unique features:

- They do not have return value declarations (not even **void**).
- They cannot be inherited, though a derived class can call the base class’s constructors and destructors.
- Constructors, like most C++ functions, can have default arguments or use member initialization lists.
- Destructors can be **virtual**, but constructors cannot. (See Virtual destructors.)
- You can’t take their addresses.

```
int main (void)
{
.
.
.
void *ptr = base::base;    // illegal
.
.
.
}
```

- Constructors and destructors can be generated by the compiler if they haven’t been explicitly defined; they are also invoked on many occasions without explicit calls in your program. Any constructor or destructor generated by the compiler will be public.

- You cannot call constructors the way you call a normal function. Destructors can be called if you use their fully qualified name.

```
{
.
.
.
X *p;
.
.
.
p->X::~X();                 // legal call of destructor
X::X();                     // illegal call of constructor
.
.
.
}
```

- The compiler automatically calls constructors and destructors when defining and destroying objects.
- Constructors and destructors can make implicit calls to operator **new** and operator **delete** if allocation is required for an object.
- An object with a constructor or destructor cannot be used as a member of a union.
- If no constructor has been defined for some class X to accept a given type, no attempt is made to find other constructors or conversion functions to convert the assigned value into a type acceptable to a constructor for class X. Note that this rule applies only to any constructor with one parameter and no initializers that use the "**=**" syntax.

```
class X { /* ... */ X(int); };
class Y { /* ... */ Y(X); };
Y a = 1;                    // illegal: Y(X(1)) not tried
```

If **class** X has one or more constructors, one of them is invoked each time you define an object x of **class** X. The constructor creates x and initializes it. Destructors reverse the process by destroying the class objects created by constructors.

Constructors are also invoked when local or temporary objects of a class are created; destructors are invoked when these objects go out of scope.

3.1.3.1.4.2 Constructors

This section contains Constructor topics.

Topics

Name	Description
Class Initialization (see page 417)	An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized. Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be the same type as the first... more (see page 417)
Constructor Defaults (see page 419)	The default constructor for **class** X is one that takes no arguments; it usually has the form X::X(). If no user-defined constructors exist for a class, the compiler generates a default constructor. On a declaration such as X x, the default constructor creates the object x. Like all functions, constructors can have default arguments. For example, the constructor
Constructors (see page 419)	Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly. Constructors for global variables are called before the main function is called. Global variable constructors are also called prior to **#pragma startup** functions. Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

Order Of Calling Constructors (see page 420)	In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared. For example, in this setup,
Overloading Constructors (see page 421)	Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.
The Copy Constructor (see page 421)	A copy constructor for **class** X is one that can be called with a single argument of type X as follows:

3.1.3.1.4.2.1 Class Initialization

An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized.

Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor. An alternative is to use an equal sign followed by a single value. The single value can be the same type as the first argument accepted by a constructor of that class, in which case either there are no additional arguments, or the remaining arguments have default values. It could also be an object of that class type. In the former case, the matching constructor is called to create the object. In the latter case, the copy constructor is called to initialize the object.

```
class X
{
   int i;
public:
   X();          // function bodies omitted for clarity
   X(int x);
   X(const X&);
};
void main()
{
   X one;         // default constructor invoked
   X two(1);      // constructor X::X(int) is used
   X three = 1;   // calls X::X(int)
   X four = one; // invokes X::X(const X&) for copy
   X five(two);   // calls X::X(const X&)
}
```

The constructor can assign values to its members in two ways:

1. It can accept the values as parameters and make assignments to the member variables within the function body of the constructor:

```
class X
{
   int a, b;
public:
   X(int i, int j) { a = i; b = j }
};
```

2. An initializer list can be used prior to the function body:

```
class X
{
   int a, b, &c;  // Note the reference variable.
public:
   X(int i, int j) : a(i), b(j), c(a) {}
};
```

The initializer list is the only place to initialize a reference variable.

In both cases, an initialization of X x(1, 2) assigns a value of 1 to x::a and 2 to x::b. The second method, the initializer list, provides a mechanism for passing values along to base class constructors.

Note: Base class constructors must be declared as either public

or **protected** to be called from a derived class.

```
class base1
{
   int x;
public:
   base1(int i) { x = i; }
};
class base2
{
   int x;
public:
   base2(int i) : x(i) {}
};
class top : public base1, public base2
{
   int a, b;
public:
   top(int i, int j) : base1(i*5), base2(j+i), a(i) { b = j;}
};
```

With this class hierarchy, a declaration of top one(1, 2) would result in the initialization of base1 with the value 5 and base2 with the value 3. The methods of initialization can be intermixed.

As described previously, the base classes are initialized in declaration order. Then the members are initialized, also in declaration order, independent of the initialization list.

```
class X
{
  int a, b;
public:
  X(int i, j) :  a(i), b(a+j) {}
};
```

With this class, a declaration of X x(1,1) results in an assignment of 1 to x::a and 2 to x::b.

Base class constructors are called prior to the construction of any of the derived classes members. If the values of the derived class are changed, they will have no effect on the creation of the base class.

```
class base
{
   int x;
public:
   base(int i) : x(i) {}
};
class derived : base
{
   int a;
public:
   derived(int i) : a(i*10), base(a) { } // Watch out! Base will be
                                         // passed an uninitialized 'a'
};
```

With this class setup, a call of derived d(1) will not result in a value of 10 for the base class member x. The value passed to the base class constructor will be undefined.

When you want an initializer list in a non-inline constructor, don't place the list in the class definition. Instead, put it at the point at which the function is defined.

```
derived::derived(int i) : a(i)
{
  .
  .
  .
}
```

See Also

Invoking Destructors (see page 424)

Atexit (see page 426)

Exit And Destructors (see page 423)

abort And Destructors (see page 425)

Virtual Destructors (see page 424)

3.1.3.1.4.2.2 Constructor Defaults

The default constructor for **class** X is one that takes no arguments; it usually has the form X::X(). If no user-defined constructors exist for a class, the compiler generates a default constructor. On a declaration such as X x, the default constructor creates the object x.

Like all functions, constructors can have default arguments. For example, the constructor

```
X::X(int, int = 0)
```

can take one or two arguments. When presented with one argument, the missing second argument is assumed to be a zero **int**. Similarly, the constructor

```
X::X(int = 5, int = 6)
```

could take two, one, or no arguments, with appropriate defaults. However, the default constructor X::X() takes no arguments and must not be confused with, say, X::X(int = 0), which can be called with no arguments as a default constructor, or can take an argument.

You should avoid ambiguity in defining constructors. In the following case, the two default constructors are ambiguous:

```
class X
{
public:
   X();
   X(int i = 0);
};
int main()
{
   X one(10);  // OK; uses X::X(int)
   X two;      // Error;ambiguous whether to call X::X() or
               // X::X(int = 0)
   return 0;
}
```

See Also

Constructors (see page 419)

3.1.3.1.4.2.3 Constructors

Constructors are distinguished from all other member functions by having the same name as the class they belong to. When an object of that class is created or is being copied, the appropriate constructor is called implicitly.

Constructors for global variables are called before the main function is called. Global variable constructors are also called prior to **#pragma startup** functions.

Local objects are created as the scope of the variable becomes active. A constructor is also invoked when a temporary object of the class is created.

```
class X {
public:
```

```
   X();   // class X constructor
};
```

A **class** X constructor cannot take X as an argument:

```
class X {
public:
   X(X);                    // illegal
};
```

The parameters to the constructor can be of any type except that of the class it's a member of. The constructor can accept a reference to its own class as a parameter; when it does so, it is called the copy constructor. A constructor that accepts no parameters is called the default constructor.

3.1.3.1.4.2.4 Order Of Calling Constructors

In the case where a class has one or more base classes, the base class constructors are invoked before the derived class constructor. The base class constructors are called in the order they are declared.

For example, in this setup,

```
class Y {...}
class X : public Y {...}
X one;
```

the constructors are called in this order:

```
Y();   // base class constructor
X();   // derived class constructor
```

For the case of multiple base classes,

```
class X : public Y, public Z
X one;
```

the constructors are called in the order of declaration:

```
Y();  // base class constructors come first
Z();
X();
```

Constructors for virtual base classes are invoked before any nonvirtual base classes. If the hierarchy contains multiple virtual base classes, the virtual base class constructors are invoked in the order in which they were declared. Any nonvirtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a nonvirtual base, that nonvirtual base will be first so that the virtual base class can be properly constructed. The code:

```
class X : public Y, virtual public Z
X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // nonvirtual base class
X();   // derived class
```

Or, for a more complicated example:

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;
```

The construction order of view would be as follows:

```
base();      // virtual base class highest in hierarchy
             // base is constructed only once
base2();     // nonvirtual base of virtual base level2
             // must be called to construct level2
level2();    // virtual base class
base2();     // nonvirtual base of level1
level1();    // other nonvirtual base
toplevel();
```

If a class hierarchy contains multiple instances of a virtual base class, that base class is constructed only once. If, however, there exist both virtual and nonvirtual instances of the base class, the class constructor is invoked a single time for all virtual instances and then once for each nonvirtual occurrence of the base class.

Constructors for elements of an array are called in increasing order of the subscript.

See Also

Class Initialization (see page 417)

Invoking Destructors (see page 424)

Atexit (see page 426)

Exit And Destructors (see page 423)

abort And Destructors (see page 425)

Virtual Destructors (see page 424)

3.1.3.1.4.2.5 Overloading Constructors

Constructors can be overloaded, allowing objects to be created, depending on the values being used for initialization.

```
class X {
   int    integer_part;
   double double_part;
public:
   X(int i)    { integer_part = i; }
   X(double d) { double_part = d; }
};
int main()
{
   X one(10);   // invokes X::X(int) and sets integer_part to 10
   X one(3.14); // invokes X::X(double) setting double_part to 3.14
   return 0;
}
```

See Also

Class Initialization (see page 417)

Invoking Destructors (see page 424)

Atexit (see page 426)

Exit And Destructors (see page 423)

abort And Destructors (see page 425)

Virtual Destructors (see page 424)

3.1.3.1.4.2.6 The Copy Constructor

A copy constructor for **class** X is one that can be called with a single argument of type X as follows:

```
X::X(X&)
```

or

```
X::X(const X&)
```

or

```
X::X(const X&, int = 0)
```

Default arguments are also allowed in a copy constructor. Copy constructors are invoked when initializing a class object, typically when you declare with initialization by another class object:

```
X x1;
X x2 = x1;
X x3(x1);
```

The compiler generates a copy constructor for **class** X if one is needed and no other constructor has been defined in **class** X. The copy constructor that is generated by the compiler lets you safely start programming with simple data types. You need to make your own definition of the copy constructor if your program creates aggregate, complex types such as **class**, **struct**, and array types. The copy constructor is also called when you pass a class argument by value to a function.

See also the discussion of member-by-member class assignment. You should define the copy constructor if you overload the assignment operator.

See Also

Constructor Defaults (see page 419)

Class Initialization (see page 417)

Invoking Destructors (see page 424)

Atexit (see page 426)

Exit And Destructors (see page 423)

abort And Destructors (see page 425)

Virtual Destructors (see page 424)

Example Of Overloading Operators (see page 439)

3.1.3.1.4.3 Destructors

This section contains Destructor topics.

Topics

Name	Description
Destructors (see page 423)	The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.
Exit And Destructors (see page 423)	When you call exit from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.
Invoking Destructors (see page 424)	A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after the main function. When pointers to objects go out of scope, a destructor is not implicitly called. This means that the **delete** operator must be called to destroy such an object. Destructors are called in the exact opposite order from which their corresponding constructors were called (see Order of calling constructors).
Virtual Destructors (see page 424)	A destructor can be declared as **virtual**. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a **virtual** destructor is itself **virtual**.

abort And Destructors (see page 425)	If you call **abort** anywhere in a program, no destructors are called, not even for variables with a global scope. A destructor can also be invoked explicitly in one of two ways: indirectly through a call to **delete**, or directly by using the destructor's fully qualified name. You can use **delete** to destroy objects that have been allocated using **new**. Explicit calls to the destructor are necessary only for objects allocated a specific address through calls to **new**
atexit, #pragma exit, And Destructors (see page 426)	All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in the main function, are destroyed as they go out of scope. The order of execution at the end of a program is as follows: • atexit() functions are executed in the order they were inserted. • **#pragma exit** functions are executed in the order of their priority codes. • Destructors for global variables are called.

3.1.3.1.4.3.1 Destructors

The destructor for a class is called to free members of an object before the object is itself destroyed. The destructor is a member function whose name is that of the class preceded by a tilde (~). A destructor cannot accept any parameters, nor will it have a return type or value declared.

```
#include <stdlib.h>
class X
{
public:
   ~X(){};  // destructor for class X
};
```

If a destructor isn't explicitly defined for a class, the compiler generates one.

See Also

Class Initialization (see page 417)

Invoking Destructors (see page 424)

Atexit (see page 426)

Exit And Destructors (see page 423)

abort And Destructors (see page 425)

Virtual Destructors (see page 424)

3.1.3.1.4.3.2 Exit And Destructors

When you call exit from within a program, destructors are not called for any local variables in the current scope. Global variables are destroyed in their normal order.

See Also

Class Initialization (see page 417)

Invoking Destructors (see page 424)

Atexit (see page 426)

abort And Destructors (see page 425)

Virtual Destructors (see page 424)

3.1.3.1.4.3.3 Invoking Destructors

A destructor is called implicitly when a variable goes out of its declared scope. Destructors for local variables are called when the block they are declared in is no longer active. In the case of global variables, destructors are called as part of the exit procedure after the main function.

When pointers to objects go out of scope, a destructor is not implicitly called. This means that the **delete** operator must be called to destroy such an object.

Destructors are called in the exact opposite order from which their corresponding constructors were called (see Order of calling constructors).

See Also

Class Initialization (see page 417)

Atexit (see page 426)

Exit And Destructors (see page 423)

abort And Destructors (see page 425)

Virtual Destructors (see page 424)

3.1.3.1.4.3.4 Virtual Destructors

A destructor can be declared as **virtual**. This allows a pointer to a base class object to call the correct destructor in the event that the pointer actually refers to a derived class object. The destructor of a class derived from a class with a **virtual** destructor is itself **virtual**.

```
/* How virtual affects the order of destructor calls.
   Without a virtual destructor in the base class, the derived
   class destructor won't be called. */
#include <iostream>
class color {
public:
   virtual ~color() {  // Virtual destructor
      std::cout << "color dtor\n";
      }
};
class red : public color {
public:
   ~red() {  // This destructor is also virtual
      std::cout << "red dtor\n";
      }
};
class brightred : public red {
public:
   ~brightred() {  // This destructor is also virtual
       std::cout << "brightred dtor\n";
       }
};
int main()
{
   color *palette[3];
   palette[0] = new red;
   palette[1] = new brightred;
   palette[2] = new color;
   // The destructors for red and color are called.
   delete palette[0];
   std::cout << std::endl;
   // The destructors for bright red, red, and color are called.
   delete palette[1];
   std::cout << std::endl;
```

```
   // The destructor for color is called.
   delete palette[2];
   return 0;
}
```

Program Output:

```
red dtor
color dtor
brightred dtor
red dtor
color dtor
color dtor
```

However, if no destructors are declared as virtual, **delete** palette[0], **delete** palette[1], and **delete** palette[2] would all call only the destructor for class color. This would incorrectly destruct the first two elements, which were actually of type red and brightred.

See Also

Class Initialization (see page 417)

Invoking Destructors (see page 424)

Atexit (see page 426)

Exit And Destructors (see page 423)

abort And Destructors (see page 425)

3.1.3.1.4.3.5 abort And Destructors

If you call **abort** anywhere in a program, no destructors are called, not even for variables with a global scope.

A destructor can also be invoked explicitly in one of two ways: indirectly through a call to **delete**, or directly by using the destructor's fully qualified name. You can use **delete** to destroy objects that have been allocated using **new**. Explicit calls to the destructor are necessary only for objects allocated a specific address through calls to **new**

```
#include <stdlib.h>
class X {
public:
   .
   .
   .
    ~X(){};
   .
   .
   .
};
void* operator new(size_t size, void *ptr)
{
   return ptr;
}
char buffer[sizeof(X)];
void main()
{
   X* pointer = new X;
   X* exact_pointer;
   exact_pointer = new(&buffer) X; // pointer initialized at
                                   // address of buffer
.
.
.
   delete pointer;                 // delete used to destroy pointer
   exact_pointer->X::~X();         // direct call used to deallocate
}
```

See Also

Class Initialization (see page 417)

Invoking Destructors (see page 424)

Atexit (see page 426)

Exit And Destructors (see page 423)

Virtual Destructors (see page 424)

3.1.3.1.4.3.6 atexit, #pragma exit, And Destructors

All global objects are active until the code in all exit procedures has executed. Local variables, including those declared in the main function, are destroyed as they go out of scope. The order of execution at the end of a program is as follows:

- atexit() functions are executed in the order they were inserted.
- **#pragma exit** functions are executed in the order of their priority codes.
- Destructors for global variables are called.

See Also

Class Initialization (see page 417)

Invoking Destructors (see page 424)

Exit And Destructors (see page 423)

abort And Destructors (see page 425)

Virtual Destructors (see page 424)

3.1.3.1.5 C++ namespaces

This section contains C++ name space topics.

Topics

Name	Description
Declaring A namespace (see page 427)	An original namespace declaration should use an identifier that has not been previously used as a global identifier.
Accessing Elements Of A namespace (see page 427)	There are three ways to access the elements of a namespace: by explicit access qualification, the using-declaration, or the using-directive. Remember that no matter which namespace you add to your local scope, identifiers in global scope (global scope is just another namespace) are still accessible by using the scope resolution operator **::**. • Explicit access qualification (see page 428) • Using directive (see page 429) • Using declaration (see page 572) **Accessing namespaces in classes** You cannot use a **using** directive inside a class. However, the **using** declarative is allowed and can be quite useful.
Anonymous namespaces (see page 427)	The C++ grammar allows you to define anonymous namespaces. To do this, you use the keyword **namespace** with no identifier before the enclosing brace.
Defining A namespace (see page 428)	The grammar for defining a namespace is

Explicit Access Qualification (see page 428)	You can explicitly qualify each member of a namespace. To do so, you use the namespace identifier together with the **::** scope resolution operator followed by the member name. For example, to access a specific member of namespace ALPHA, you write:
Extending A namespace (see page 428)	Namespaces are discontinuous and open for additional development. If you redeclare a namespace, the effect is that you extend the original namespace by adding new declarations. Any extensions that are made to a namespace after a using-declaration, will not be known at the point at which the using-declaration occurs. Therefore, all overloaded versions of some function should be included in the namespace before you declare the function to be in use.
namespace Alias (see page 429)	You can use an alternate name to refer to a namespace identifier. An alias is useful when you need to refer to a long, unwieldy namespace identifier.
using Directive (see page 429)	If you want to use several (or all of) the members of a namespace, C++ provides an easy way to get access to the complete namespace. The using-directive causes all identifiers in a namespace to be in scope at the point that the using-directive statement is made. The grammar for the using-directive is as follows. using-directive: **using namespace ::** opt nested-name-specifier opt namespace-name; The using-directive is transitive. When you apply the using directive to a namespace that contains using directives within itself, you get access to those namespaces as well. For example, if you apply the using directive in your... more (see page 429)

3.1.3.1.5.1 Declaring A namespace

An original namespace declaration should use an identifier that has not been previously used as a global identifier.

```
namespace alpha {  /* ALPHA is the identifier of this namespace. */
  /* your program declarations */
  long double LD;
  float f(float y) { return y; }
  }
```

A namespace identifier must be known in all translation units where you intend to access its elements.

See Also

namespace Alias (see page 429)

3.1.3.1.5.2 Accessing Elements Of A namespace

There are three ways to access the elements of a namespace: by explicit access qualification, the using-declaration, or the using-directive. Remember that no matter which namespace you add to your local scope, identifiers in global scope (global scope is just another namespace) are still accessible by using the scope resolution operator **::**.

- Explicit access qualification (see page 428)
- Using directive (see page 429)
- Using declaration (see page 572)

Accessing namespaces in classes

You cannot use a **using** directive inside a class. However, the **using** declarative is allowed and can be quite useful.

See Also

using Directive (see page 429)

using (declaration) (see page 572)

3.1.3.1.5.3 Anonymous namespaces

The C++ grammar allows you to define anonymous namespaces. To do this, you use the keyword **namespace** with no identifier before the enclosing brace.

```
namespace {          // Anonymous namespace
   // Declarations
   }
```

All anonymous, unnamed namespaces in global scope (that is, unnamed namespaces that are not nested) of the same translation unit share the same namespace. This way you can make static declarations without using the **static** keyword.

Each identifier that is enclosed within an unnamed namespace is unique within the translation unit in which the unnamed namespace is defined.

3.1.3.1.5.4 Defining A namespace

The grammar for defining a namespace is

```
original-namespace-name:
    identifier
 namespace-definition:
    original-namespace-definition
    extension-namespace-definition
    unnamed-namespace-definition
```

Grammatically, there are three ways to define a namespace with the **namespace** keyword:

```
original-namespace-definition:
    namespace identifier { namespace-body }
extension-namespace-definition:
    namespace original-namespace-name { namespace-body }
unnamed-namespace-definition:
    namespace { namespace-body }
```

The body is an optional sequence of declarations. The grammar is

```
namespace-body:
    declaration-seq opt
```

See Also

Declaring A namespace (see page 427)

Extending A namespace (see page 428)

3.1.3.1.5.5 Explicit Access Qualification

You can explicitly qualify each member of a namespace. To do so, you use the namespace identifier together with the **::** scope resolution operator followed by the member name. For example, to access a specific member of namespace ALPHA, you write:

```
ALPHA::LD;  // Access a variable
ALPHA::f;   // Access a function
```

Explicit access qualification can always be used to resolve ambiguity. No matter which namespace (except anonymous namespace) is being used in your subsystem, you can apply the scope resolution operator **::** to access identifiers in any namespace (including a namespace already being used in the local scope) or the global namespace. Therefore, any identifier in the application can be accessed with sufficient qualification.

New-style typecasting

This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier cast expressions (which are still available) in the C language.

Types cannot be defined in a cast.

3.1.3.1.5.6 Extending A namespace

Namespaces are discontinuous and open for additional development. If you redeclare a namespace, the effect is that you extend

the original namespace by adding new declarations. Any extensions that are made to a namespace after a using-declaration, will not be known at the point at which the using-declaration occurs. Therefore, all overloaded versions of some function should be included in the namespace before you declare the function to be in use.

3.1.3.1.5.7 namespace Alias

You can use an alternate name to refer to a namespace identifier. An alias is useful when you need to refer to a long, unwieldy namespace identifier.

```
namespace BORLAND_SOFTWARE_CORPORATION {
   /* namespace-body */
   namespace NESTED_BORLAND_SOFTWARE_CORPORATION {
      /* namespace-body */
      }
   }
// Alias namespace
namespace BI = BORLAND_SOFTWARE_CORPORATION;
// Use access qualifier to alias a nested namespace
namespace NBI = BORLAND_SOFTWARE_CORPORATION::NESTED_BORLAND_SOFTWARE_CORPORATION;
```

3.1.3.1.5.8 using Directive

If you want to use several (or all of) the members of a namespace, C++ provides an easy way to get access to the complete namespace. The using-directive causes all identifiers in a namespace to be in scope at the point that the using-directive statement is made. The grammar for the using-directive is as follows.

using-directive:

using namespace :: opt nested-name-specifier opt namespace-name;

The using-directive is transitive. When you apply the using directive to a namespace that contains using directives within itself, you get access to those namespaces as well. For example, if you apply the using directive in your program, you also get namespaces `qux`, `foo`, and `bar`.

```
namespace qux {
   using namespace foo;  // This has been defined previously
   using namespace bar;  // This also has been defined previously
   }
```

The using-directive does not add any identifiers to your local scope. Therefore, an identifier defined in more than one namespace won't be a problem until you actually attempt to use it. Local scope declarations take precedence by hiding all other similar declarations.

Warning: Do not use the using

directive in header files. You might accidentally break namespaces in client code.

See Also

Accessing Elements Of A namespace (see page 427)

using (declaration) (see page 572)

3.1.3.1.6 C++ Scope

This section contains C++ Scope topics.

Topics

Name	Description
C++ Scope (see page 430)	The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement might appear. The latter flexibility means that care is needed when interpreting such phrases as “enclosing scope” and “point of declaration.”
C++ Scoping Rules Summary (see page 431)	The following rules apply to all names, including **typedef** names and class names, provided that C++ allows such names in the particular context discussed: • The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated. • If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested. • If the name is used outside any function and class, or is prefixed by the unary scope access operator ::, and if the name is not qualified by the binary :: operator or the member selection... more (see page 431)
Class Scope (see page 431)	The name M of a member of a class X has class scope “local to X”; it can be used only in the following situations: • In member functions of X • In expressions such as x.M, where x is an object of X • In expressions such as xptr->M, where xptr is a pointer to an object of X • In expressions such as X::M or D::M, where D is a derived class of X • In forward references within the class of which it is a member Names of functions declared as friends of X are not members of X; their names simply... more (see page 431)
Hiding (see page 432)	A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: X::M. A hidden file scope (global) name can be referenced with the unary operator :: (for example, ::g). A class name X can be hidden by the name of an object, function, or enumerator declared within the scope of X, regardless of the order in which the names are declared. However, the hidden class name X can still be accessed by prefixing X with... more (see page 432)

3.1.3.1.6.1 C++ Scope

The lexical scoping rules for C++, apart from class scope, follow the general rules for C, with the proviso that C++, unlike C, permits both data and function declarations to appear wherever a statement might appear. The latter flexibility means that care is needed when interpreting such phrases as “enclosing scope” and “point of declaration.”

See Also

Class Scope (see page 431)

Hiding (see page 432)

C++ Scoping Rules Summary (see page 431)

3.1.3.1.6.2 C++ Scoping Rules Summary

The following rules apply to all names, including **typedef** names and class names, provided that C++ allows such names in the particular context discussed:

- The name itself is tested for ambiguity. If no ambiguities are detected within its scope, the access sequence is initiated.
- If no access control errors occur, the type of the object, function, class, **typedef**, and so on, is tested.
- If the name is used outside any function and class, or is prefixed by the unary scope access operator ::, and if the name is not qualified by the binary :: operator or the member selection operators **.** and **->**, then the name must be a global object, function, or enumerator.
- If the name n appears in any of the forms X::n, x.n (where x is an object of X or a reference to X), or ptr->n (where ptr is a pointer to X), then n is the name of a member of X or the member of a class from which X is derived.
- Any name that hasn't been discussed yet and that is used in a static member function must either be declared in the block it occurs in or in an enclosing block, or be a global name. The declaration of a local name n hides declarations of n in enclosing blocks and global declarations of n. Names in different scopes are not overloaded.
- Any name that hasn't been discussed yet and that is used in a nonstatic member function of class X must either be declared in the block it occurs in or in an enclosing block, be a member of class X or a base class of X, or be a global name. The declaration of a local name n hides declarations of n in enclosing blocks, members of the function's class, and global declarations of n. The declaration of a member name hides declarations of the same name in base classes.
- The name of a function argument in a function definition is in the scope of the outermost block of the function. The name of a function argument in a nondefining function declaration has no scope at all. The scope of a default argument is determined by the point of declaration of its argument, but it can't access local variables or nonstatic class members. Default arguments are evaluated at each point of call.
- A constructor initializer is evaluated in the scope of the outermost block of its constructor, so it can refer to the constructor's argument names.

See Also

C++ Scope (see page 430)

Class Scope (see page 431)

Hiding (see page 432)

3.1.3.1.6.3 Class Scope

The name M of a member of a class X has class scope "local to X"; it can be used only in the following situations:

- In member functions of X
- In expressions such as x.M, where x is an object of X
- In expressions such as xptr->M, where xptr is a pointer to an object of X
- In expressions such as X::M or D::M, where D is a derived class of X
- In forward references within the class of which it is a member

Names of functions declared as friends of X are not members of X; their names simply have enclosing scope.

See Also

C++ Scope (see page 430)

Hiding (see page 432)

C++ Scoping Rules Summary (see page 431)

3.1.3.1.6.4 Hiding

A name can be hidden by an explicit declaration of the same name in an enclosed block or in a class. A hidden class member is still accessible using the scope modifier with a class name: X::M. A hidden file scope (global) name can be referenced with the unary operator :: (for example, ::g). A class name X can be hidden by the name of an object, function, or enumerator declared within the scope of X, regardless of the order in which the names are declared. However, the hidden class name X can still be accessed by prefixing X with the appropriate keyword: **class**, **struct**, or **union**.

The point of declaration for a name x is immediately after its complete declaration but before its initializer, if one exists.

See Also

C++ Scope (see page 430)

Class Scope (see page 431)

C++ Scoping Rules Summary (see page 431)

3.1.3.1.7 Exporting And Importing Templates

This section contains Exporting And Importing Template topics.

Topics

Name	Description
Exporting And Importing Templates (see page 432)	The declaration of a template function or template class needs to be sufficiently flexible to allow it to be used in either a dynamic link library (shared library) or an executable file. The same template declaration should be available as an import and/or export, or without a modifier. To be completely flexible, the header file template declarations should not use __**export** or __**import** modifiers. This allows the user to apply the appropriate modifier at the point of instantiation depending on how the instantiation is to be used. The following steps demonstrate exporting and importing of templates. The source code is... more (see page 432)

3.1.3.1.7.1 Exporting And Importing Templates

The declaration of a template function or template class needs to be sufficiently flexible to allow it to be used in either a dynamic link library (shared library) or an executable file. The same template declaration should be available as an import and/or export, or without a modifier. To be completely flexible, the header file template declarations should not use __**export** or __**import** modifiers. This allows the user to apply the appropriate modifier at the point of instantiation depending on how the instantiation is to be used.

The following steps demonstrate exporting and importing of templates. The source code is organized in three files. Using the header file, code is generated in the dynamic link library.

1. Exportable/Importable Template Declarations

The header file contains all template class and template function declarations. An export/import version of the templates can be instantiated by defining the appropriate macro at compile time.

2. Compiling Exportable Templates

Write the source code for a dynamic link library. When compiled, this library has reusable export code for templates.

3. Using ImportTemplates

Now you can write a calling function that uses templates. This executable file is linked to the dynamic link library. Only objects that are not declared in the header file and which are instantiated in the main function cause the compiler to generate new code. Code for a newly instantiated object is written into main.obj file.

3.1.3.1.8 Function Templates Overview

This section contains Function Templates Overview topics.

Topics

Name	Description
Function Templates (see page 433)	Consider a function max(x, y) that returns the larger of its two arguments. x and y can be of any type that has the ability to be ordered. But, since C++ is a strongly typed language, it expects the types of the parameters x and y to be declared at compile time. Without using templates, many overloaded versions of max are required, one for each data type to be supported even though the code for each version is essentially identical. Each version compares the arguments and returns the larger. One way around this problem is to use a macro:
Implicit And Explicit Template Functions (see page 434)	When doing overload resolution (following the steps of looking for an exact match), the compiler ignores template functions that have been generated implicitly by the compiler.
Overriding A Template Function (see page 434)	The previous example is called a function template (or generic function, if you like). A specific instantiation of a function template is called a template function. Template function instantiation occurs when you take the function address, or when you call the function with defined (non-generic) data types. You can override the generation of a template function for a specific type with a non-template function:

3.1.3.1.8.1 Function Templates

Consider a function max(x, y) that returns the larger of its two arguments. x and y can be of any type that has the ability to be ordered. But, since C++ is a strongly typed language, it expects the types of the parameters x and y to be declared at compile time. Without using templates, many overloaded versions of max are required, one for each data type to be supported even though the code for each version is essentially identical. Each version compares the arguments and returns the larger.

One way around this problem is to use a macro:

```
#define max(x,y) ((x > y) ? x : y)
```

However, using the #**define** circumvents the type-checking mechanism that makes C++ such an improvement over C. In fact, this use of macros is almost obsolete in C++. Clearly, the intent of max(x, y) is to compare compatible types. Unfortunately, using the macro allows a comparison between an **int** and a **struct**, which are incompatible.

Another problem with the macro approach is that substitution will be performed where you don't want it to be. By using a template instead, you can define a pattern for a family of related overloaded functions by letting the data type itself be a parameter:

```
template <class T> T max(T x, T y){
   return (x > y) ? x : y;
   };
```

The data type is represented by the template argument **<class T>**. When used in an application, the compiler generates the appropriate code for the max function according to the data type actually used in the call:

```
int i;
Myclass a, b;
int j = max(i,0);       // arguments are integers
Myclass m = max(a,b);   // arguments are type Myclass
```

Any data type (not just a class) can be used for **<class T>**. The compiler takes care of calling the appropriate **operator>()**, so you can use max with arguments of any type for which **operator>()** is defined.

See Also

Exporting And Importing Templates (see page 432)

Implicit And Explicit Template Functions (see page 434)

3.1.3.1.8.2 Implicit And Explicit Template Functions

When doing overload resolution (following the steps of looking for an exact match), the compiler ignores template functions that have been generated implicitly by the compiler.

```
template<class T> T max(T a, T b){
        return  (a > b) ? a : b;
}
void f(int i, char c)
{
    max(i, i);      // calls max(int ,int )
    max(c, c);      // calls max(char,char)
    max(i, c);      // no match for max(int,char)
    max(c, i);      // no match for max(char,int)
}
```

This code results in the following error messages:

```
Could not find a match for 'max(int,char)' in function f(int,char)Could not find a match for
'max(char,int)' in function f(int,char)
Could not find a match for 'max(char,int)' in function f(int,char)
```

If the user explicitly declares a function, this function, on the other hand, will participate fully in overload resolution. See the example of explicit function.

When searching for an exact match for template function parameters, trivial conversions are considered to be exact matches. See the example on trivial conversions.

Template functions with derived class pointer or reference arguments are permitted to match their public base classes. See the example of base class referencing.

3.1.3.1.8.3 Overriding A Template Function

The previous example is called a function template (or generic function, if you like). A specific instantiation of a function template is called a template function. Template function instantiation occurs when you take the function address, or when you call the function with defined (non-generic) data types. You can override the generation of a template function for a specific type with a non-template function:

```
#include <string.h>
char *max(char *x, char *y){
   return(strcmp(x,y) > 0) ? x : y;
}
```

If you call the function with string arguments, it's executed in place of the automatic template function. In this case, calling the function avoided a meaningless comparison between two pointers.

Only trivial argument conversions are performed with compiler-generated template functions.

The argument type(s) of a template function must use all of the template formal arguments. If it doesn't, there is no way of deducing the actual values for the unused template arguments when the function is called.

3.1.3.1.9 The new And delete Operators

This section contains new And delete Operator topics.

Topics

Name	Description
Handling Errors For The New Operator (see page 435)	By default, **new** throws the bad_alloc exception when a request for memory allocation cannot be satisfied. You can define a function to be called if the **new** operator fails. To tell the **new** operator about the new-handler function, use set_new_handler and supply a pointer to the new-handler. If you want **new** to return null on failure, you must use set_new_handler(0) .
Operator new Placement Syntax (see page 436)	The placement syntax for **operator new( )** can be used only if you have overloaded the allocation operator with the appropriate arguments. You can use the placement syntax when you want to use and reuse a memory space which you set up once at the beginning of your program. When you use the overloaded **operator new( )** to specify where you want an allocation to be placed, you are responsible for deleting the allocation. Because you call your version of the allocation operator, you cannot depend on the global **::operator delete( )** to do the cleanup. To release memory, you... more (see page 436)
Overloading The Operator delete (see page 436)	The global operators, ::operator delete(), and **::operator delete[]()** cannot be overloaded. However, you can override the default version of each of these operators with your own implementation. Only one instance of the each global delete function can exist in the program. The user-defined operator delete must have a **void** return type and **void*** as its first argument; a second argument of type size_t is optional. A class T can define at most one version of each of T**::operator delete[]()** and T**::operator delete()**. To overload the **delete** operators, use the following prototypes. • void operator delete(void *Type_ptr, [size_t... more (see page 436)
Overloading The Operator new (see page 436)	The global ::operator new() and **::operator new[]()** can be overloaded. Each overloaded instance must have a unique signature. Therefore, multiple instances of a global allocation operator can coexist in a single program. Class-specific memory allocation operators can also be overloaded. The operator **new** can be implemented to provide alternative free storage (heap) memory-management routines, or implemented to accept additional arguments. A user-defined operator **new** must return a **void*** and must have a size_t as its first argument. To overload the **new** operators, use the following prototypes declared in the new.h header file. • void * operator new(size_t Type_size); // For... more (see page 436)
The delete Operator With Arrays (see page 437)	Arrays are deleted by operator delete[](). You must use the syntax **delete []** expr when deleting an array.
Operator new (see page 437)	By default, if there is no overloaded version of new, a request for dynamic memory allocation always uses the global version of new, **::operator new()**. A request for array allocation calls **::operator new[]()**. With class objects of type name, a specific operator called name**::operator new()** or name**::operator new[]()** can be defined. When **new** is applied to class name objects it invokes the appropriate name**::operator new** if it is present; otherwise, the global **::operator new** is used. Only the **operator new()** function will accept an optional initializer. The array allocator version, **operator new[](),** will not accept... more (see page 437)
The Operator new With Arrays (see page 437)	When using the array form of operator new[](), the pointer returned points to the first element of the array. When creating multidimensional arrays with **new**, all array sizes must be supplied (although the leftmost dimension doesn't have to be a compile-time constant):

3.1.3.1.9.1 Handling Errors For The New Operator

By default, **new** throws the bad_alloc exception when a request for memory allocation cannot be satisfied.

You can define a function to be called if the **new** operator fails. To tell the **new** operator about the new-handler function, use set_new_handler and supply a pointer to the new-handler. If you want **new** to return null on failure, you must use set_new_handler(0) .

See Also

new (see page 556)

3.1.3.1.9.2 Operator new Placement Syntax

The placement syntax for **operator new( )** can be used only if you have overloaded the allocation operator with the appropriate arguments. You can use the placement syntax when you want to use and reuse a memory space which you set up once at the beginning of your program.

When you use the overloaded **operator new( )** to specify where you want an allocation to be placed, you are responsible for deleting the allocation. Because you call your version of the allocation operator, you cannot depend on the global **::operator delete( )** to do the cleanup.

To release memory, you make an explicit call on the destructor. This method for cleaning up memory should be used only in special situations and with great care. If you make an explicit call of a destructor before an object that has been constructed on the stack goes out of scope, the destructor will be called again when the stackframe is cleaned up.

See Also

The keyword new (see page 556)

3.1.3.1.9.3 Overloading The Operator delete

The global operators, ::operator delete(), and **::operator delete[]()** cannot be overloaded. However, you can override the default version of each of these operators with your own implementation. Only one instance of the each global delete function can exist in the program.

The user-defined operator delete must have a **void** return type and **void*** as its first argument; a second argument of type size_t is optional. A class T can define at most one version of each of T**::operator delete[]()** and T**::operator delete()**. To overload the **delete** operators, use the following prototypes.

- void operator delete(void *Type_ptr, [size_t Type_size]); // For Non-array
- void operator delete[](size_t Type_ptr, [size_t Type_size]); // For arrays

See Also

The keyword delete (see page 546)

3.1.3.1.9.4 Overloading The Operator new

The global ::operator new() and **::operator new[]()** can be overloaded. Each overloaded instance must have a unique signature. Therefore, multiple instances of a global allocation operator can coexist in a single program.

Class-specific memory allocation operators can also be overloaded. The operator **new** can be implemented to provide alternative free storage (heap) memory-management routines, or implemented to accept additional arguments. A user-defined operator **new** must return a **void*** and must have a size_t as its first argument. To overload the **new** operators, use the following prototypes declared in the new.h header file.

- void * operator new(size_t Type_size); // For Non-array
- void * operator new[](size_t Type_size); // For arrays

The compiler provides Type_size to the **new** operator. Any data type may be substituted for Type_size except function names (although a pointer to function is permitted), class declarations, enumeration declarations, const, volatile.

See Also

The keyword new (see page 556)

3.1.3.1.9.5 The delete Operator With Arrays

Arrays are deleted by operator delete[](). You must use the syntax **delete []** expr when deleting an array.

```
char * p;
void func()
{
   p = new char[10];    //  allocate 10 chars
   delete[] p;          // delete 10 chars
}
```

Early C++ compilers required the array size to be named in the delete expression. In order to handle legacy code, the compiler issues a warning and simply ignores any size that is specified. For example, if the preceding example reads delete[10] p and is compiled, the warning is as follows:

```
Warning: Array size for 'delete' ignored in function func()
```

See Also

The keyword delete (see page 546)

3.1.3.1.9.6 Operator new

By default, if there is no overloaded version of new, a request for dynamic memory allocation always uses the global version of new, **::operator new()**. A request for array allocation calls **::operator new[]()**. With class objects of type name, a specific operator called name**::operator new()** or name**::operator new[]()** can be defined. When **new** is applied to class name objects it invokes the appropriate name**::operator new** if it is present; otherwise, the global **::operator new** is used.

Only the **operator new()** function will accept an optional initializer. The array allocator version, **operator new[](),** will not accept initializers. In the absence of explicit initializers, the object created by new contains unpredictable data (garbage). The objects allocated by **new**, other than arrays, can be initialized with a suitable expression between parentheses:

```
int_ptr = new int(3);
```

Arrays of classes with constructors are initialized with the default constructor. The user-defined **new** operator with customized initialization plays a key role in C++ constructors for class-type objects.

See Also

Overloading The Operator New (see page 436)

The keyword new (see page 556)

3.1.3.1.9.7 The Operator new With Arrays

When using the array form of operator new[](), the pointer returned points to the first element of the array. When creating multidimensional arrays with **new**, all array sizes must be supplied (although the leftmost dimension doesn't have to be a compile-time constant):

```
mat_ptr = new int[3][10][12];    // OK
mat_ptr = new int[n][10][12];    // OK
mat_ptr = new int[3][][12];      // illegal
mat_ptr = new int[][10][12];     // illegal
```

Although the first array dimension can be a variable, all following dimensions must be constants.

See Also

The keyword new (see page 556)

3.1.3.1.10 New-style Typecasting Overview

This section contains New-style Typecasting Overview topics.

Topics

Name	Description
New-style Typecasting (see page 438)	This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier cast expressions (which are still available) in the C language. Types cannot be defined in a cast.

3.1.3.1.10.1 New-style Typecasting

This section presents a discussion of alternate methods for making a typecast. The methods presented here augment the earlier cast expressions (which are still available) in the C language.

Types cannot be defined in a cast.

See Also

Const_cast (typecast Operator) (see page 539)

Dynamic_cast (typecast Operator) (see page 547)

Reinterpret_cast (typecast Operator) (see page 561)

static_cast (typecast Operator) (see page 565)

3.1.3.1.11 Operator Overloading Overview

This section contains Operator Overloading Overview topics.

Topics

Name	Description
How To Construct A Class Of Complex Vectors (see page 438)	This section contains How to Construct A Class Of Complex Vector topics.
Overloading Operators (see page 441)	C++ lets you redefine the actions of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands. All the operators can be overloaded except for:

3.1.3.1.11.1 How To Construct A Class Of Complex Vectors

This section contains How to Construct A Class Of Complex Vector topics.

Topics

Name	Description
Example Of Overloading Operators (see page 439)	The following example extends the class complex to create complex-type vectors. Several of the most useful operators are overloaded to provide some customary mathematical operations in the usual mathematical syntax. Some of the issues illustrated by the example are: • The default constructor is defined. The default constructor is provided by the compiler only if you have not defined it or any other constructor. • The copy constructor is defined explicitly. Normally, if you have not defined any constructors, the compiler will provide one. You should define the copy constructor if you are overloading the assignment operator. • The assignment operator is overloaded.... more (see page 439)

3.1.3.1.11.1.1 Example Of Overloading Operators

The following example extends the class complex to create complex-type vectors. Several of the most useful operators are overloaded to provide some customary mathematical operations in the usual mathematical syntax.

Some of the issues illustrated by the example are:

- The default constructor is defined. The default constructor is provided by the compiler only if you have not defined it or any other constructor.
- The copy constructor is defined explicitly. Normally, if you have not defined any constructors, the compiler will provide one. You should define the copy constructor if you are overloading the assignment operator.
- The assignment operator is overloaded. If you do not overload the assignment operator, the compiler calls a default assignment operator when required. By overloading assignment of cvector types, you specify exactly the actions to be taken. Note that derived classes cannot inherit the assignment operator.
- The subscript operator is defined as a member function (a requirement when overloading) with a single argument. The **const** version assures the caller that it will not modify its argument—this is useful when copying or assigning. This operator should check that the index value is within range—a good place to implement exception handling.
- The addition operator is defined as a member function. It allows addition only for cvector types. Addition should always check that the operands' sizes are compatible.
- The multiplication operator is declared a **friend**. This lets you define the order of the operands. An attempt to reverse the order of the operands is a compile-time error.
- The stream insertion operator is overloaded to naturally display a cvector. Large objects that don't display well on a limited size screen might require a different display strategy.

Example Source

```
/* HOW TO EXTEND THE complex CLASS AND OVERLOAD THE REQUIRED OPERATORS. */
complexcomplexcomplex
#include <complex>  // This includes iostream
using namespace std;
// COMPLEX VECTORS
template <class T>
class cvector {
   int size;
   complex<T> *data;
public:
   cvector() { size = 0; data = NULL; };
   cvector(int i = 5) : size(i) {    // DEFAULT VECTOR SIZE.
      data = new complex<T>[size];
      for (int j = 0; j < size; j++)
           data[j] = j + (0.1 * j);  // ARBITRARY INITIALIZATION.
```

```
    };
  /* THIS VERSION IS CALLED IN main() */
  complex<T>& operator [](int i) { return data[i]; };
  /* THIS VERSION IS CALLED IN ASSIGNMENT OPERATOR AND COPY THE CONSTRUCTOR */
  const complex<T>& operator [](int i) const { return data[i]; };
  cvector operator +(cvector& A) {  // ADDITION OPERATOR
     cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
     for (int i = 0; i < size; i++)
         result[i] = data[i] + A.data[i];
     return result;
     };
  /* BECAUSE scalar * vector MULTIPLICATION IS NOT COMMUTATIVE, THE ORDER OF
     THE ELEMENTS MUST BE SPECIFIED. THIS FRIEND OPERATOR FUNCTION WILL ENSURE
     PROPER MULTIPLICATION. */
  friend cvector operator *(T scalar, cvector& A) {
     cvector result(A.size);  // DO NOT MODIFY THE ORIGINAL
     for (int i = 0; i < A.size; i++)
       result.data[i] = scalar * A.data[i];
     return result;
     }
  /* THE STREAM INSERTION OPERATOR. */
  friend ostream& operator <<(ostream& out_data, cvector& C) {
     for (int i = 0; i < C.size; i++)
          out_data << "[" << i << "]=" << C.data[i] << "   ";
     cout << endl;
     return out_data;
     };
  cvector( const cvector &C ) {  // COPY CONSTRUCTOR
     size = C.size;
     data = new complex<T>[size];
     for (int i = 0; i < size; i++)
          data[i] = C[i];
    }
  cvector& operator =(const cvector &C) { // ASSIGNMENT OPERATOR.
     if (this == &C) return *this;
     delete[] data;
     size = C.size;
     data = new complex<T>[size];
     for (int i = 0; i < size; i++)
         data[i] = C[i];
     return *this;
  };
  virtual ~cvector() { delete[] data; }; // DESTRUCTOR
  };
int main(void) { /* A FEW OPERATIONS WITH complex VECTORS. */
  cvector<float> cvector1(4), cvector2(4), result(4);
  // CREATE complex NUMBERS AND ASSIGN THEM TO complex VECTORS
  cvector1[3] = complex<float>(3.3, 102.8);
  cout << "Here is cvector1:" << endl;
  cout << cvector1;
  cvector2[3] = complex<float>(33.3, 81);
  cout << "Here is cvector2:" << endl;
  cout << cvector2;
  result = cvector1 + cvector2;
  cout << "The result of vector addition:" << endl;
  cout << result;
  result = 10 * cvector2;
  cout << "The result of 10 * cvector2:" << endl;
  cout << result;
  return 0;
  }
```

Output

```
Here is cvector1:
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(3.3, 102.8)
Here is cvector2:
```

```
[0]=(0, 0)   [1]=(1.1, 0)   [2]=(2.2, 0)   [3]=(33.3, 81)
The result of vector addition:
[0]=(0, 0)   [1]=(2.2, 0)   [2]=(4.4, 0)   [3]=(36.6, 183.8)
The result of 10 * cvector2:
[0]=(0, 0)   [1]=(11, 0)   [2]=(22, 0)   [3]=(333, 810)
```

3.1.3.1.11.2 Overloading Operators

C++ lets you redefine the actions of most operators, so that they perform specified functions when used with objects of a particular class. As with overloaded C++ functions in general, the compiler distinguishes the different functions by noting the context of the call: the number and types of the arguments or operands.

All the operators can be overloaded except for:

```
.   .*   ::   ?:
```

The following preprocessing symbols cannot be overloaded.

```
#   ##
```

The =, [ ], ( ), and -> operators can be overloaded only as nonstatic member functions. These operators cannot be overloaded for **enum** types. Any attempt to overload a global version of these operators results in a compile-time error.

The keyword **operator** followed by the operator symbol is called the operator function name; it is used like a normal function name when defining the new (overloaded) action for the operator.

A function operator called with arguments behaves like an operator working on its operands in an expression. The operator function cannot alter the number of arguments or the precedence and associativity rules applying to normal operator use.

See Also

Example Of Overloading Operators (see page 439)

Overloaded Operators And Inheritance (see page 444)

Overloading Binary Operators (see page 442)

Overloading Operator Functions (see page 444)

Overloading The Assignment Operator = (see page 442)

Overloading The Class Member Access Operators -> (see page 443)

Overloading The Function Call Operator ( ) (see page 443)

Overloading The Subscript Operator [] (see page 445)

3.1.3.1.12 Overloading Operator Functions Overview

This section contains Overloading Operator Functions Overview topics.

Topics

Name	Description
Overloading The Assignment operator = (see page 442)	The assignment operator=( ) can be overloaded by declaring a nonstatic member function. For example,
Overloading Binary Operators (see page 442)	You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member function (usually friend) taking two arguments. If @ represents a binary operator, x@y can be interpreted as either x.operator@(y) or operator@(x,y) depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.
Overloading The Class Member Access Operators -> (see page 443)	**Syntax**
Overloading The Function Call Operator ( ) (see page 443)	**Syntax**

Overloading Operator Functions (see page 444)	Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:
Overloaded Operators And Inheritance (see page 444)	With the exception of the assignment function operator =( ), all overloaded operator functions for class X are inherited by classes derived from X, with the standard resolution rules for overloaded functions. If X is a base class for Y, an overloaded operator function for X could possibly be further overloaded for Y.
Overloading The Subscript Operator [] (see page 445)	**Syntax**
Overloading Unary Operators (see page 445)	You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a nonmember function taking one argument. If @ represents a unary operator, @x and x@ can both be interpreted as either x.operator@() or operator@(x), depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity. • Under C++ 2.0, an overloaded operator ++ or -- is used for both prefix and postfix uses of the operator. • With C++ 2.1, when an operator ++ or operator -- is declared as a member... more (see page 445)

3.1.3.1.12.1 Overloading The Assignment operator =

The assignment operator=( ) can be overloaded by declaring a nonstatic member function. For example,

```
class String {
      .
      .
      .
   String& operator = (String& str);
      .
      .
      .
   String (String&);
   ~String();
}
```

This code, with suitable definitions of String::operator =(), allows string assignments str1 = str2 in the usual sense. Unlike the other operator functions, the assignment operator function cannot be inherited by derived classes. If, for any class X, there is no user-defined operator =, the operator = is defined by default as a member-by-member assignment of the members of class X:

```
X& X::operator = (const X& source)
{
   // memberwise assignment
}
```

See Also

Example Of Overloading Operators (see page 439)

Overloaded Operators And Inheritance (see page 444)

Overloading Binary Operators (see page 442)

Overloading Operator Functions (see page 444)

Overloading The Class Member Access Operators -> (see page 443)

Overloading The Function Call Operator ( ) (see page 443)

Overloading The Subscript Operator [] (see page 445)

3.1.3.1.12.2 Overloading Binary Operators

You can overload a binary operator by declaring a nonstatic member function taking one argument, or by declaring a non-member function (usually friend) taking two arguments. If @ represents a binary operator, x@y can be interpreted as either

x.operator@(y) or operator@(x,y) depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

See Also

Example Of Overloading Operators (see page 439)

Overloaded Operators And Inheritance (see page 444)

Overloading Operator Functions (see page 444)

Overloading The Assignment Operator = (see page 442)

Overloading The Class Member Access Operators -> (see page 443)

Overloading The Function Call Operator ( ) (see page 443)

Overloading The Subscript Operator [] (see page 445)

3.1.3.1.12.3 Overloading The Class Member Access Operators ->

Syntax

```
postfix-expression -> primary-expression
```

Description

The expression x->m, where x is a **class** X object, is interpreted as (x.operator->())->m, so that the function **operator->()** must either return a pointer to a class object or return an object of a class for which **operator->** is defined.

The **operator->()** can only be overloaded as a nonstatic member function.

See Also

Example Of Overloading Operators (see page 439)

Overloaded Operators And Inheritance (see page 444)

Overloading Binary Operators (see page 442)

Overloading Operator Functions (see page 444)

Overloading The Assignment Operator = (see page 442)

Overloading The Function Call Operator ( ) (see page 443)

Overloading The Subscript Operator [] (see page 445)

3.1.3.1.12.4 Overloading The Function Call Operator ( )

Syntax

```
postfix-expression ( <expression-list> )
```

Description

In its ordinary use as a function call, the postfix-expression must be a function name, or a pointer or reference to a function. When the postfix-expression is used to make a member function call, postfix-expression must be a class member function name or a pointer-to-member expression used to select a class member function. In either case, the postfix-expression is followed by the optional expression-list.

A call X(arg1, arg2), where X is an object class X, is interpreted as X.**operator()**(arg1, arg2).

The function call operator, **operator()()**, can only be overloaded as a nonstatic member function.

See Also

Example Of Overloading Operators (see page 439)

Overloaded Operators And Inheritance (see page 444)

Overloading Binary Operators (see page 442)

Overloading Operator Functions (see page 444)

Overloading The Assignment Operator = (see page 442)

Overloading The Class Member Access Operators -> (see page 443)

Overloading The Subscript Operator [] (see page 445)

3.1.3.1.12.5 Overloading Operator Functions

Operator functions can be called directly, although they are usually invoked indirectly by the use of the overload operator:

```
c3 = c1.operator + (c2);   // same as c3 = c1 + c2
```

Apart from new and delete, which have their own rules, an operator function must either be a nonstatic member function or have at least one argument of class type. The operator functions =, ( ), [ ] and -> must be nonstatic member functions.

Enumerations can have overloaded operators. However, the operator functions =, ( ), [ ], and -> cannot be overloaded for enumerations.

See Also

Example Of Overloading Operators (see page 439)

Overloaded Operators And Inheritance (see page 444)

Overloading Binary Operators (see page 442)

Overloading The Assignment Operator = (see page 442)

Overloading The Class Member Access Operators -> (see page 443)

Overloading The Function Call Operator ( ) (see page 443)

Overloading The Subscript Operator [] (see page 445)

3.1.3.1.12.6 Overloaded Operators And Inheritance

With the exception of the assignment function operator =( ), all overloaded operator functions for class X are inherited by classes derived from X, with the standard resolution rules for overloaded functions. If X is a base class for Y, an overloaded operator function for X could possibly be further overloaded for Y.

See Also

Example Of Overloading Operators (see page 439)

Overloading Binary Operators (see page 442)

Overloading Operator Functions (see page 444)

Overloading The Assignment Operator = (see page 442)

Overloading The Class Member Access Operators -> (see page 443)

Overloading The Function Call Operator ( ) (see page 443)

Overloading The Subscript Operator [] (see page 445)

3.1.3.1.12.7 Overloading The Subscript Operator []

Syntax

```
postfix-expression [ expression ]
```

Description

The corresponding operator function is operator[]() this can be user-defined for a class X (and any derived classes). The expression X[y], where X is an object of class X, is interpreted as x.operator[](y).

The **operator[]()** can only be overloaded as a nonstatic member function.

See Also

Example Of Overloading Operators (see page 439)

Overloaded Operators And Inheritance (see page 444)

Overloading Binary Operators (see page 442)

Overloading Operator Functions (see page 444)

Overloading The Assignment Operator = (see page 442)

Overloading The Function Call Operator ( ) (see page 443)

Overloading The Class Member Access Operators -> (see page 443)

3.1.3.1.12.8 Overloading Unary Operators

You can overload a prefix or postfix unary operator by declaring a nonstatic member function taking no arguments, or by declaring a nonmember function taking one argument. If @ represents a unary operator, @x and x@ can both be interpreted as either x.operator@() or operator@(x), depending on the declarations made. If both forms have been declared, standard argument matching is applied to resolve any ambiguity.

- Under C++ 2.0, an overloaded operator ++ or -- is used for both prefix and postfix uses of the operator.
- With C++ 2.1, when an operator ++ or operator -- is declared as a member function with no parameters, or as a nonmember function with one parameter, it only overloads the prefix operator ++ or operator --. You can only overload a postfix operator ++ or operator -- by defining it as a member function taking an int parameter or as a nonmember function taking one class and one int parameter.

When only the prefix version of an operator ++ or operator -- is overloaded and the operator is applied to a class object as a postfix operator, the compiler issues a warning. Then it calls the prefix operator, allowing 2.0 code to compile. The preceding example results in the following warnings:

```
Warning: Overloaded prefix 'operator ++' used as a postfix operator in function func()
Warning: Overloaded prefix 'operator --' used as a postfix operator in function func()
```

See Also

Example Of Overloading Operators (see page 439)

Overloaded Operators And Inheritance (see page 444)

Overloading Binary Operators (see page 442)

Overloading Operator Functions (see page 444)

Overloading The Assignment Operator = (see page 442)

Overloading The Class Member Access Operators -> (see page 443)

Overloading The Function Call Operator ( ) (see page 443)

Overloading The Subscript Operator [] (see page 445)

3.1.3.1.13 Polymorphic Classes

This section contains Polymorphic Class topics.

Topics

Name	Description
Abstract Classes (see page 446)	This section contains Abstract Class topics.
Polymorphic Classes (see page 447)	Classes that provide an identical interface, but can be implemented to serve different specific requirements, are referred to as polymorphic classes. A class is polymorphic if it declares or inherits at least one virtual (or pure virtual) function. The only types that can support polymorphism are **class** and **struct**.
Virtual Functions (see page 447)	This section contains Virtual Function topics.

3.1.3.1.13.1 Abstract Classes

This section contains Abstract Class topics.

Topics

Name	Description
Abstract Classes (see page 446)	An abstract class is a class with at least one pure **virtual** function. A **virtual** function is specified as pure by setting it equal to zero. An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example,

3.1.3.1.13.1.1 Abstract Classes

An abstract class is a class with at least one pure **virtual** function. A **virtual** function is specified as pure by setting it equal to zero.

An abstract class can be used only as a base class for other classes. No objects of an abstract class can be created. An abstract class cannot be used as an argument type or as a function return type. However, you can declare pointers to an abstract class. References to an abstract class are allowed, provided that a temporary object is not needed in the initialization. For example,

```
class shape {       // abstract class
   point center;
.
.
.
public:
   where() { return center; }
   move(point p) { center = p; draw(); }
   virtual void rotate(int) = 0; // pure virtual function
   virtual void draw() = 0;      // pure virtual function
   virtual void hilite() = 0;    // pure virtual function
.
.
.
}
shape x;// ERROR: attempt to create an object of an abstract class
   shape* sptr;// pointer to abstract class is OK
   shape f();// ERROR: abstract class cannot be a return type
int g(shape s);// ERROR: abstract class cannot be a function argument type
shape& h(shape&);// reference to abstract class as return
 // value or function argument is OK
```

Suppose that D is a derived class with the abstract class B as its immediate base class. Then for each pure virtual function pvf in B, if D doesn't provide a definition for pvf, pvf becomes a pure member function of D, and D will also be an abstract class.

For example, using the class shape previously outlined,

```
class circle : public shape {// circle derived from abstract class
   int radius;// private
public:
   void rotate(int) { }// virtual function defined: no action
            //  to rotate a circle
   void draw();      // circle::draw must be defined somewhere
}
```

Member functions can be called from a constructor of an abstract class, but calling a pure virtual function directly or indirectly from such a constructor provokes a runtime error.

See Also

Polymorphic Classes (see page 447)

Virtual Functions (see page 449)

Dynamic Functions (see page 447)

3.1.3.1.13.2 Polymorphic Classes

Classes that provide an identical interface, but can be implemented to serve different specific requirements, are referred to as polymorphic classes. A class is polymorphic if it declares or inherits at least one virtual (or pure virtual) function. The only types that can support polymorphism are **class** and **struct**.

See Also

Virtual Functions (see page 449)

Dynamic Functions (see page 447)

Abstract Classes (see page 446)

3.1.3.1.13.3 Virtual Functions

This section contains Virtual Function topics.

Topics

Name	Description
Dynamic Functions (see page 447)	**dynamic** functions are allowed for classes derived from TObject. Dynamic functions are similar to virtual functions except for the way they are stored in the virtual tables. Virtual functions occupy a slot in the virtual table in the object they are defined in, and in the virtual table of every descendant of that object. Dynamic functions occupy a slot in every object that defines them, not in any descendants. That is, dynamic functions are virtual functions stored in sparse virtual tables. If you call a dynamic function, and that function is not defined in your object, the virtual tables of... more (see page 447)
Virtual Functions (see page 449)	**virtual** functions allow derived classes to provide different versions of a base class function. You can use the **virtual** keyword to declare a **virtual** function in a base class. By declaring the function prototype in the usual way and then prefixing the declaration with the **virtual** keyword. To declare a pure function (which automatically declares an abstract class), prefix the prototype with the **virtual** keyword, and set the function equal to zero.

3.1.3.1.13.3.1 Dynamic Functions

dynamic functions are allowed for classes derived from TObject. Dynamic functions are similar to virtual functions except for the way they are stored in the virtual tables. Virtual functions occupy a slot in the virtual table in the object they are defined in, and in the virtual table of every descendant of that object. Dynamic functions occupy a slot in every object that defines them, not in any descendants. That is, dynamic functions are virtual functions stored in sparse virtual tables. If you call a dynamic function, and that function is not defined in your object, the virtual tables of its ancestors are searched until the function is found.

Therefore, dynamic functions reduces the size of your virtual tables at the expense of a delay at runtime to look up the address of the functions.

Because dynamic functions are available only in classes derived from TObject, you will get an error if you use them in a regular class. For example:

```
class dynfunc {
int __declspec(dynamic) bar() { return 5; }
};
```

gives the syntax error, “Error: Storage class 'dynamic' is not allowed here.” But, the following code compiles.

```
#include <clxvcl.h>
#include <stdio.h>
class __declspec(delphiclass) func1 : public TObject {
public:
func1() {}
int virtual virtbar() { return 5; }
int __declspec(dynamic) dynbar() { return 5; }
};
class __declspec(delphiclass) func2 : public func1 {
public:
func2() {}
};
class __declspec(delphiclass) func3 : public func2 {
public:
func3() {}
int virtbar() { return 10; }
int dynbar() { return 10; }
};
int main()
{
func3 * Func3 = new func3;
func1 * Func1 = Func3;
printf("func3->dynbar: %d\n", Func3->dynbar());
printf("func3->virtbar: %d\n", Func3->virtbar());
printf("func1->dynbar: %d\n", Func1->dynbar());
printf("func1->virtbar: %d\n", Func1->virtbar());
delete Func3;
func2 * Func2 = new func2;
printf("func2->dynbar: %d\n", Func2->dynbar());
printf("func2->virtbar: %d\n", Func2->virtbar());
delete Func2;
return 0;
}
```

Dynamic attribute is inherited

Since dynamic functions are just like virtual functions, the dynamic attribute is automatically inherited. You can verify this by running the above example. When you generate assembly output with "bcc32 -S" you can examine the virtual tables of func1, func2, and func3, and you'll see how func2 has NO entry for dynbar, but it does have an entry for virtbar. Still, you can call dynbar in the func2 object:

Output:

```
func3->dynbar: 10
func3->virtbar: 10
func1->dynbar: 10
func1->virtbar: 10
func2->dynbar: 5
func2->virtbar: 5
```

Dynamic functions cannot be made virtual, and vice-versa

You cannot redeclare a virtual function to be dynamic; likewise, you cannot redeclare a dynamic function to be virtual. The following example gives errors:

```
#include <clxvcl.h>
#include <stdio.h>
class __declspec(delphiclass) foo1 : public TObject {
public:
foo1() {}
int virtual virtbar() { return 5; }
int __declspec(dynamic) dynbar() { return 5; }
};
class __declspec(delphiclass) foo2 : public foo1 {
public:
foo2() {}
int __declspec(dynamic) virtbar() { return 10; }
int virtual dynbar() { return 10; }
};
Error : Cannot override a virtual with a dynamic function
Error : Cannot override a dynamic with a virtual function
```

See Also

Polymorphic Classes (see page 447)

Virtual Functions (see page 449)

Dynamic Functions

Abstract Classes (see page 446)

3.1.3.1.13.3.2 Virtual Functions

virtual functions allow derived classes to provide different versions of a base class function. You can use the **virtual** keyword to declare a **virtual** function in a base class. By declaring the function prototype in the usual way and then prefixing the declaration with the **virtual** keyword. To declare a pure function (which automatically declares an abstract class), prefix the prototype with the **virtual** keyword, and set the function equal to zero.

```
virtual int funct1(void);       // A virtual function declaration.
virtual int funct2(void) = 0;   // A pure function declaration.
```

A function declaration cannot provide both a pure-specifier and a definition.

Example

```
struct C {
    virtual void f() { } = 0; // ill-formed
};
```

The only legal syntax to provide a body is:

```
struct TheClass
{
  virtual void funct3(void) = 0;
};
virtual void TheClass::funct3(void)
{
   // Some code here.
};
```

Note: See Abstract classesAbstractClasses for a discussion of pure virtual functions.

When you declare **virtual** functions, keep these guidelines in mind:

- They can be member functions only.
- They can be declared a **friend** of another class.
- They cannot be a static member.

A **virtual** function does not need to be redefined in a derived class. You can supply one definition in the base class so that all calls will access the base function.

To redefine a **virtual** function in any derived class, the number and type of arguments must be the same in the base class declaration and in the derived class declaration. (The case for redefined **virtual** functions differing only in return type is discussed below.) A redefined function is said to override the base class function.

You can also declare the functions int Base::Fun(int) and int Derived::Fun(int) even when they are not virtual. In such a case, int Derived::Fun(int) is said to hide any other versions of Fun(int) that exist in any base classes. In addition, if class Derived defines other versions of Fun(), (that is, versions of Fun() with different signatures) such versions are said to be overloaded versions of Fun().

Virtual function return types

Generally, when redefining a **virtual** function, you cannot change just the function return type. To redefine a **virtual** function, the new definition (in some derived class) must exactly match the return type and formal parameters of the initial declaration. If two functions with the same name have different formal parameters, C++ considers them different, and the **virtual** function mechanism is ignored.

However, for certain virtual functions in a base class, their overriding version in a derived class can have a return type that is different from the overridden function. This is possible only when both of the following conditions are met:

- The overridden **virtual** function returns a pointer or reference to the base class.
- The overriding function returns a pointer or reference to the derived class.

If a base class B and class D (derived publicly from B) each contain a **virtual** function vf, then if vf is called for an object d of D, the call made is D::vf(), even when the access is via a pointer or reference to B. For example,

```
struct X {};// Base class.
struct Y : X {};// Derived class.
struct B {
   virtual void vf1();
   virtual void vf2();
   virtual void vf3();
   void f();
   virtual X* pf();// Return type is a pointer to base. This can
  //  be overridden.
};
class D : public B {
public:
   virtual void vf1();// Virtual specifier is legal but redundant.
   void vf2(int);// Not virtual, since it's using a different
 //  arg list. This hides B::vf2().
// char vf3();// Illegal: return-type-only change!
   void f();
   Y*   pf();// Overriding function differs only
   //  in return type. Returns a pointer to
 //  the derived class.
};
void extf()
{
   D d;// Instantiate D
   B* bp = &d;// Standard conversion from D* to B*
  // Initialize bp with the table of functions
// provided for object d. If there is no entry for a
  // function in the d-table, use the function
     //  in the B-table.
   bp->vf1();  // Calls D::vf1
   bp->vf2();  // Calls B::vf2 since D's vf2 has different args
   bp->f();    // Calls B::f (not virtual)
   X* xptr = bp->pf();// Calls D::pf() and converts the result
   //  to a pointer to X.
   D* dptr = &d;
   Y* yptr = dptr->pf();// Calls D::pf() and initializes yptr.
   //  No further conversion is done.
}
```

The overriding function vf1 in D is automatically **virtual**. The **virtual** specifier can be used with an overriding function declaration in the derived class. If other classes will be derived from D, the **virtual** keyword is required. If no further classes will be derived

from D, the use of **virtual** is redundant.

The interpretation of a **virtual** function call depends on the type of the object it is called for; with nonvirtual function calls, the interpretation depends only on the type of the pointer or reference denoting the object it is called for.

virtual functions exact a price for their versatility: each object in the derived class needs to carry a pointer to a table of functions in order to select the correct one at runtime (late binding).

See Also

Polymorphic Classes (see page 447)

Dynamic Functions (see page 447)

Abstract Classes (see page 446)

3.1.3.1.14 Referencing

This section contains Typeid Operator topics.

Topics

Name	Description
Reference Arguments (see page 451)	The reference declarator can also be used to declare reference type parameters within a function:
Referencing (see page 452)	In the C programming language, you can pass arguments only by value. In C++, you can pass arguments by value or by reference. C++ reference types, closely related to pointer types, create aliases for objects. See the following topics for a discussion of referencing. **Note:** C++ specific pointer referencing and dereferencing is discussed in C++ specific operators (see page 590).
Simple References (see page 453)	The reference declarator can be used to declare references outside functions:

3.1.3.1.14.1 Reference Arguments

The reference declarator can also be used to declare reference type parameters within a function:

```
void func1 (int i);
void func2 (int &ir);   // ir is type "reference to int"
    .
    .
    .
int sum = 3;
func1(sum);             // sum passed by value
func2(sum);             // sum passed by reference
```

The sum argument passed by reference can be changed directly by func2. On the other hand, func1 gets a copy of the sum argument (passed by value), so sum itself cannot be altered by func1.

When an actual argument x is passed by value, the matching formal argument in the function receives a copy of x. Any changes to this copy within the function body are not reflected in the value of x outside the scope of the function. Of course, the function can return a value that could be used later to change x, but the function cannot directly alter a parameter passed by value.

In C, changing the value of a function parameter outside the scope of the function requires that you pass the address of the parameter. The address is passed by value, thus changing the contents of the address effects the value of the parameter outside the scope of the function.

Even if the function does not need to change the value of a parameter, it is still useful to pass the address (or a reference) to a function. This is especially true if the parameter is a large data structure or object. Passing an object directly to a function necessitates copying the entire object.

Compare the three implementations of the function treble:

Implementation 1

```
int treble_1(int n)
{
   return 3 * n;
}
  .
  .
  .
int x, i = 4;
x = treble_1(i);  // x now = 12, i = 4
  .
  .
  .
```

Implementation 2

```
void treble_2(int* np)
{
   *np = (*np) * 3;
}
  .
  .
  .
treble_2(&i);    // i now = 2
```

Implementation 3

```
void treble_3(int& n)   // n is a reference type
{
   n = n * 3;
}
  .
  .
  .
treble_3(i);             // i now = 36
```

The formal argument declaration **type**& t establishes t as type “reference to type.” So, when treble_3 is called with the real argument i, i is used to initialize the formal reference argument n. n therefore acts as an alias for i, so n = n*3 also assigns 3 * i to i.

If the initializer is a constant or an object of a different type than the reference type, creates a temporary object for which the reference acts as an alias:

```
int& ir = 6;   /* temporary int object created, aliased by ir, gets value 6 */
float f;
int& ir2 = f;  /* creates temporary int object aliased by ir2; f converted
                  before assignment */
ir2 = 2.0      // ir2 now = 2, but f is unchanged
```

The automatic creation of temporary objects permits the conversion of reference types when formal and actual arguments have different (but assignment-compatible) types. When passing by value, of course, there are fewer conversion problems, since the copy of the actual argument can be physically changed before assignment to the formal argument.

See Also

Referencing (see page 452)

Simple References (see page 453)

3.1.3.1.14.2 Referencing

In the C programming language, you can pass arguments only by value. In C++, you can pass arguments by value or by reference. C++ reference types, closely related to pointer types, create aliases for objects. See the following topics for a discussion of referencing.

Note: C++ specific pointer referencing and dereferencing is discussed in C++ specific operators (see page 590).

See Also

Simple References (see page 453)

Reference Arguments (see page 451)

Reference/Dereference Operators (see page 658)

3.1.3.1.14.3 Simple References

The reference declarator can be used to declare references outside functions:

```
int  i  = 0;
int &ir = i;   // ir is an alias for i
ir = 2;        // same effect as i = 2
```

This creates the lvalue ir as an alias for i, provided the initializer is the same type as the reference. Any operations on ir have precisely the same effect as operations on i. For example, ir = 2 assigns 2 to i, and &ir returns the address of i.

See Also

Referencing (see page 452)

Reference Arguments (see page 451)

3.1.3.1.15 Run-time Type Identification (RTTI)

This section contains Run-time Type Identification (RTTI) topics.

Topics

Name	Description
Runtime Type Identification (RTTI) Overview (see page 453)	Runtime type identification (RTTI) lets you write portable code that can determine the actual type of a data object at runtime even when the code has access only to a pointer or reference to that object. This makes it possible, for example, to convert a pointer to a virtual base class into a pointer to the derived type of the actual object. Use the dynamic_cast operator to make runtime casts. The RTTI mechanism also lets you check whether an object is of some particular type and whether two objects are of the same type. You can do this with typeid... more (see page 453)
The Typeid Operator (see page 454)	This section contains Typeid Operator topics.

3.1.3.1.15.1 Runtime Type Identification (RTTI) Overview

Runtime type identification (RTTI) lets you write portable code that can determine the actual type of a data object at runtime even when the code has access only to a pointer or reference to that object. This makes it possible, for example, to convert a pointer to a virtual base class into a pointer to the derived type of the actual object. Use the dynamic_cast operator to make runtime casts.

The RTTI mechanism also lets you check whether an object is of some particular type and whether two objects are of the same type. You can do this with typeid operator, which determines the actual type of its argument and returns a reference to an object of type **const** type_info, which describes that type.

You can also use a type name as the argument to **typeid**, and **typeid** will return a reference to a **const** type_info object for that type. The class type_info provides an **operator==** and an **operator!=** that you can use to determine whether two objects are of the same type. Class type_info also provides a member function name that returns a pointer to a character string that holds the name of the type.

See Also

__rtti (see page 528)

The Typeid Operator (see page 454)

3.1.3.1.15.2 The Typeid Operator

This section contains Typeid Operator topics.

Topics

Name	Description
Runtime Type Identification And Destructors (see page 454)	When destructor cleanup is enabled, a pointer to a class with a virtual destructor can't be deleted if that class is not compiled with runtime type identification enabled. The runtime type identification and destructor cleanup options are on by default. They can be disabled from the C++ page of the Project Options dialog box, or by using the **-xd-** and **-RT-** command-line options. **Example**

3.1.3.1.15.2.1 Runtime Type Identification And Destructors

When destructor cleanup is enabled, a pointer to a class with a virtual destructor can't be deleted if that class is not compiled with runtime type identification enabled. The runtime type identification and destructor cleanup options are on by default. They can be disabled from the C++ page of the Project Options dialog box, or by using the **-xd-** and **-RT-** command-line options.

Example

```
class Alpha {
public:
   virtual ~Alpha( ) { }
};
void func( Alpha *Aptr ) {
   delete Aptr;         // Error.  Alpha is not a polymorphic class type
}
```

3.1.3.1.16 The Scope Resolution Operator

This section contains Scope Resolution Operator topics.

Topics

Name	Description
Scope Resolution Operator :: (see page 454)	The scope access (or resolution) operator **::** (two colons) lets you access a global (or file duration) member name even if it is hidden by a local redeclaration of that name. You can use a global identifier by prefixing it with the scope resolution operator. You can access a nested member name by specifying the class name and using the scope resolution operator. Therefore, Alpha::func( ) and Beta::func( ) are two different functions.

3.1.3.1.16.1 Scope Resolution Operator ::

The scope access (or resolution) operator **::** (two colons) lets you access a global (or file duration) member name even if it is hidden by a local redeclaration of that name. You can use a global identifier by prefixing it with the scope resolution operator. You can access a nested member name by specifying the class name and using the scope resolution operator. Therefore, Alpha::func( ) and Beta::func( ) are two different functions.

3.1.3.1.17 Stricter C++ Compiler (C++Builder 2007)

To more closely obey the rules of the C++ ANSI Standard, the C++ compiler shipping with C++Builder 2007 is stricter than previous versions. Code that did not generate errors in earlier versions of C++Builder might fail to compile beginning with

C++Builder 2007.

This section lists some of the common areas where the compiler is stricter. Each case is illustrated with an example showing the problem and how to update the code to compile with C++Builder 2007. Note that there are often many ways to bring offending code up to date. The appropriate method depends on the intent of the original code.

Topics

Name	Description
Stricter C++ Compiler: Binding of References and Qualifiers (see page 455)	There are many constructs that now generate error messages from the CodeGear C++ compiler included with C++Builder 2007 and newer releases. The rules governing this behavior are described in section 8.5.3 of the 2003 C++ ANSI standard. The rules can be divided into the following categories (with the compiler switch that overrides this behavior): • Binding a non-const lvalue to to non-const reference. (Use the compiler switch **-Vbr** to allow this.) • Binding a temporary to a non-const reference. (Use the compiler switch **-Vbr** to allow this.) • Binding of const or volatile objects to non-const or non-volatile methods repectively. (Use the compiler... more (see page 455)
Stricter C++ Compiler: String Literals Are Now Constants (see page 457)	String literals are now considered to be of type 'const char[]' by default. This, combined with the stricter qualification binding of const values and types, can generate error messages in code that compiled before. You might enable the **-Vbs** switch to revert string literals to non-const. However, CodeGear recommends that you update the code instead. Note that the change in the type of string literals can also change how the compiler resolves calls to overloaded methods. The following example illustrates this:
Stricter C++ Compiler: Template Changes (see page 457)	The C++ compiler no longer allows an explicit template without the 'template <>' prefix. Use the compiler switch **-Vbe** to allow this. The following example shows this:
Stricter C++ Compiler: Function Overload Resolution (see page 458)	One of the areas where the C++Builder 2007 compiler differs the most from the previous versions is in overload resolution, which includes the detection of ambiguity. The compiler now better conforms to the rules in section 13.3 of the 2003 C++ ANSI Standard. Several constructs that were previously allowed might now be reported as ambiguous or no match found, requiring that you modify code to clarify its intent. The compiler option **-Vbo** reverts to the old behavior, not enforcing the new stricter behavior. However, not all compiler changes can be controlled by this switch, so CodeGear recommends that you update... more (see page 458)
Stricter C++ Compiler: Initialization and Conversion (see page 459)	The compiler now obeys the rules of 8.5.1 and 13.3.1 of the 2003 C++ ANSI Standard for initialization and conversion: • Direct initialization now requires initialization by a constructor and no longer picks a user conversion sequence. • Copy initialization for objects of the same or derived type now requires a constructor call. • Copy initialization for objects of the different types no longer prefers user conversion over construction. If the compiler finds a suitable user conversion, it now continues to look for (possibly ambiguous) converting constructors. If the chosen conversion function is a converting constructor, the call initializes a temporary of the... more (see page 459)

3.1.3.1.17.1 Stricter C++ Compiler: Binding of References and Qualifiers

There are many constructs that now generate error messages from the CodeGear C++ compiler included with C++Builder 2007

and newer releases. The rules governing this behavior are described in section 8.5.3 of the 2003 C++ ANSI standard.

The rules can be divided into the following categories (with the compiler switch that overrides this behavior):

- Binding a non-const lvalue to to non-const reference. (Use the compiler switch **-Vbr** to allow this.)
- Binding a temporary to a non-const reference. (Use the compiler switch **-Vbr** to allow this.)
- Binding of const or volatile objects to non-const or non-volatile methods repectively. (Use the compiler switch **-Vbn** to allow this.)

Previous versions of the Borland C++ compilers allowed various forms of binding to non-const reference parameters. In the following example, for instance, one was allowed to simply cast the **psize** parameter:

```
int takesLongRef(long& l);

  int takesUnsignedPtr(unsigned long* psize) {

     return takesLongRef((long)*psize);
  }
```

With C++Builder2007, the above code generates these errors:

```
Error E2357 test.cpp 3: Reference initialized with 'long', needs lvalue of type 'long' in
function takesUnsignedPtr(unsigned long *)
Error E2342 test.cpp 3: Type mismatch in parameter 'l' (wanted 'long &', got 'long') in
function takesUnsignedPtr(unsigned long *)
```

To remedy this, you can cast **psize** before dereferencing, as in:

```
int takesLongRef(long& l);
int takesUnsignedPtr(unsigned long* psize) {     return takesLongRef(*reinterpret_cast
<long*>(psize));   }
```

Be aware of cases that involve temporaries in unobvious ways. For example, some binary operators imply a temporary:

```
enum { zero, one, two } num;
num |= two; // Not allowed
num = num | two; // OK
```

Another case that involves temporaries in an unobvious way is the return value of a property. The following example illustrates code that compiled with previous versions of the compiler:

```
#include <vcl.h>

class TTest {
   WideString FData ;
public:
   __property WideString Data = {read = FData };
};

void Func(WideString& wref);

void test() {
   TTest t;
   Func(t.Data);
}
```

With C++Builder 2007, the above generates two errors:

```
Error E2357 test.cpp 14: Reference initialized with 'const WideString', needs lvalue of type
'WideString' in function test()
Error E2342 test.cpp 14: Type mismatch in parameter 'wref' (wanted 'WideString &', got
'WideString') in function test()
```

You can fix this by changing the reference to a **const** reference, as in:

```
void Func(const WideString& wref);
```

Here is an example of trying to bind a const object to a non-const method:

```
struct X {
   void foo();
};
const X x;
x.foo(); //error
```

Reconcile Error Dialog: Solving Problems

The temporaries and references issue referred to above is encountered in code generated by previous versions of the Reconcile Error Dialog Wizard.

To remedy this, look for the **VarToAnsiStr** method:

```
AnsiString VarToAnsiStr (Variant &V TFieldType DataType)
```

and change it to take a **const Variant&**, as in:

```
AnsiString VarToAnsiStr (const Variant &V, TFieldType DataType)
```

See Also

Stricter C++ Compiler (C++Builder 2007) (see page 454)

String Literals Are Now Constants (see page 457)

Template Changes (see page 457)

Function Overload Resolution (see page 458)

Initialization and Conversion (see page 459)

3.1.3.1.17.2 Stricter C++ Compiler: String Literals Are Now Constants

String literals are now considered to be of type 'const char[]' by default. This, combined with the stricter qualification binding of const values and types, can generate error messages in code that compiled before.

You might enable the **-Vbs** switch to revert string literals to non-const. However, CodeGear recommends that you update the code instead.

Note that the change in the type of string literals can also change how the compiler resolves calls to overloaded methods. The following example illustrates this:

```
void foo(char *);
 void foo(const char *);
 foo("string"); // New Compiler picks foo(const char *)
```

With C++Builder2007, the above code generates these errors:

See Also

Binding of References and Qualifiers (see page 455)

Template Changes (see page 457)

Function Overload Resolution (see page 458)

Initialization and Conversion (see page 459)

3.1.3.1.17.3 Stricter C++ Compiler: Template Changes

The C++ compiler no longer allows an explicit template without the 'template <>' prefix. Use the compiler switch **-Vbe** to allow this. The following example shows this:

```
template <class>
class foo {
 foo();
```

```
};
foo<int>::foo();//Error
template<> foo<int>::foo();//OK
```

Also, the C++ compiler no longer allows explicit template specialization within a class. Use the compiler switch **-Vbx** to allow this. For example, the following generates an error:

```
struct S {};
  struct SP
  {
    template <typename> void foo(const T &) {}
    template <> void foo(const S &) {} // Error
  };

  template <> void SP::foo(const S &) {} //OK
```

See Also

Stricter C++ Compiler (C++Builder 2007) (see page 454)

Binding of References and Qualifiers (see page 455)

String Literals Are Now Constants (see page 457)

Function Overload Resolution (see page 458)

Initialization and Conversion (see page 459)

3.1.3.1.17.4 Stricter C++ Compiler: Function Overload Resolution

One of the areas where the C++Builder 2007 compiler differs the most from the previous versions is in overload resolution, which includes the detection of ambiguity. The compiler now better conforms to the rules in section 13.3 of the 2003 C++ ANSI Standard. Several constructs that were previously allowed might now be reported as ambiguous or no match found, requiring that you modify code to clarify its intent.

The compiler option **-Vbo** reverts to the old behavior, not enforcing the new stricter behavior. However, not all compiler changes can be controlled by this switch, so CodeGear recommends that you update the code instead.

The following is an example of an ambiguity that was permitted by the previous compiler:

```
class X{};
  void foo(X);
  void foo(const X&);
  void ambig() {
    X x;
    foo(x); //error-ambiguous-the previous compiler chose 'void foo(x)'
  }
```

std::abs Ambiguity

The standard **abs** function might also generate an ambiguity message when invoked with a parameter that does not exactly match the types expected by the various overloaded versions of abs. Here is an example:

```
#include <limits>bool test(long l) {   return std::abs(l)> 0;}
```

The code above generates an error and a warning:

```
Error E2015 test.cpp 5: Ambiguity between 'std::abs(int) at C:\dev\tp\sc\include\math.h:208'
and 'std::abs(long double) at C:\dev\tp\sc\include\math.h:275' in function test(long)
Warning W8057 test.cpp 6: Parameter 'l' is never used in function test(long)
```

To fix this, cast to the type of the overload you want to invoke. For example:

```
  #include <limits>bool test(long l) {   return std::abs(static_cast<int>(l)) > 0;}
```

See Also

Stricter C++ Compiler (C++Builder 2007) (see page 454)

Binding of References and Qualifiers (see page 455)

String Literals Are Now Constants (see page 457)

Template Changes (see page 457)

Initialization and Conversion (see page 459)

3.1.3.1.17.5 Stricter C++ Compiler: Initialization and Conversion

The compiler now obeys the rules of 8.5.1 and 13.3.1 of the 2003 C++ ANSI Standard for initialization and conversion:

- Direct initialization now requires initialization by a constructor and no longer picks a user conversion sequence.
- Copy initialization for objects of the same or derived type now requires a constructor call.
- Copy initialization for objects of the different types no longer prefers user conversion over construction. If the compiler finds a suitable user conversion, it now continues to look for (possibly ambiguous) converting constructors. If the chosen conversion function is a converting constructor, the call initializes a temporary of the destination type. The result of the call (which is the temporary for the constructor case) is then used to directinitialize the object. Use the compiler switch **-Vbo** to revert to the previous behavior.
- For an explicit cast, the compiler now performs direct initialization on a temporary.

This example illustrates the new behavior:

```
// In this example, dst is destination type and src is source type    class A { };    class V
{    public:      V() { };      V( const V & ) { }      V( const A & ) { }    };G g; V v;
  // direct initialization
  // ==> constructors are considered.
  V v9(g);
  // Both of these statements previously compiled but now get the error:
  // Error E2015: Ambiguity between 'V::V(const V &)' and 'V::V(const A &)'

  // casts
  // (V)g is treated as V tmp(g) which is direct initialization of 'tmp'
  // ==> constructors are considered.
  (V)g;
  static_cast<V> (g);
  // Both of these statements previously compiled but now get the error:
  // Error E2015: Ambiguity between 'V::V(const V &)' and 'V::V(const A &)'

  // copy initialization with dst=V src=G
  // ==> user-defined conversion sequences are considered.
  V v4 = g;
  V v5 = G();
  // Both of these statements now compile but previously got the error:
  // Error E2015: Ambiguity between 'V::V(const A &)' and 'V::V(const V &)'

  // copy initialization with dst=V src=V
  // ==> converting constructors of V are considered.
  V v6 = (V)g;
  V v7 = V(g);
  // Both of these statements compiled previously but now get the error:
  // Error E2015: Ambiguity between 'V::V(const V &)' and 'V::V(const A &)'
```

Conversion via User-Defined Operators

The new C++Builder 2007 compiler often reports ambiguities for conversions that involve user-defined operators. An example is shown below:

```
class AnsiString   {   public:     bool operator ==(const AnsiString& other);
AnsiString(const wchar_t* src);   };
```

```
class Variant   {   public:     operator AnsiString() const;operator wchar_t*() const;
bool operator ==(const AnsiString& rhs) const     { return static_cast<AnsiString>(*this) ==
rhs;}
```

C++Builder users might notice that the above is a stripped down version of the VCL AnsiString and Variant classes. Previous versions of the compiler invoked the 'Variant' 'operator AnsiString() const' for 'static_cast<AnsiString>(*this)', while C++Builder 2007 uses 'conversion via constructor'. Since the Variant can be converted to multiple types for which there are AnsiString constructors, the compiler generates an ambiguity error.

To correct this ambiguity error, you must eliminate the cast as in:

```
 bool operator ==(const AnsiString& rhs) const
  { return (*this) == rhs;}
```

You can also be explicit about the operator:

```
bool operator ==(const AnsiString& rhs) const
  { return this->operator AnsiString() == rhs; }
```

Variant/OleVariant/AnsiString/WideString/TDateTime

The issue described above with a user-defined conversion operator vs. conversion via constructor might be encountered in several constructs involving the VCL classes Variant, OleVariant, AnsiString, WideString, TDateTime, Currency, and so forth.

The following table lists constructs that now generate error messages and the updated syntax.

Previous Construct	Updated Construct	Notes
AnsiString test(OleVariant v) { AnsiString ret = (AnsiString) v; return ret; }	AnsiString test(OleVariant v) { AnsiString ret = /*(AnsiString)*/ v; return ret; }	Do not cast RHS when relying on conversion operator in an assignment.
WideString test(OleVariant v) { WideString w(v); return w; }	WideString test(OleVariant v) { WideString w = v; return w; }	Use Copy Initialization instead of the more direct constructor.

The underlying compiler change for the errors described above is related to the way the compiler now handles initialization and conversion.

See Also

Stricter C++ Compiler (C++Builder 2007) (see page 454)

Binding of References and Qualifiers (see page 455)

String Literals Are Now Constants (see page 457)

Template Changes (see page 457)

Function Overload Resolution (see page 458)

3.1.3.1.18 Templates

This section contains Template topics.

Topics

Name	Description
Template Body Parsing (see page 461)	Earlier versions of the compiler didn't check the syntax of a template body unless the template was instantiated. A template body is now parsed immediately when seen like every other declaration.

Using Templates (see page 462)	Templates, also called generics or parameterized types, let you construct a family of related functions or classes. These topics introduce the basic concept of templates: Exporting and importing templates (see page 432) Template Body Parsing (see page 461) Function Templates (see page 433) Class Templates (see page 412) Implicit and Explicit Template Functions (see page 434) Template Compiler Switches (see page 414) The Keyword template (see page 568)

3.1.3.1.18.1 Template Body Parsing

Earlier versions of the compiler didn't check the syntax of a template body unless the template was instantiated. A template body is now parsed immediately when seen like every other declaration.

```
template <class T> class X : T
{
  Int  j;  // Error: Type name expected in template X<T>
};
```

Let's assume that Int hasn't yet been defined. This means that Int must be a member of the template argument T. But it also might just be a typing error and should be **int** instead of Int. Because the compiler can't guess the right meaning it issues an error message.

If you want to access types defined by a template argument you should use a **typedef** to make your intention clear to the compiler:

```
template <class T> class X : T
{
  typedef  typename T::Int  Int;
  Int  j;
};
```

You cannot just write

```
   typedef   T::Int;
```

as in earlier versions of the compiler. Not giving the **typedef** name was acceptable, but this now causes an error message.

All other templates mentioned inside the template body are declared or defined at that point. Therefore, the following example is ill-formed and will not compile:

```
template <class T> class  X
{
  void f(NotYetDefindedTemplate<T> x);
};
```

All template definitions must end with a semicolon. Earlier versions of the compiler did not complain if the semicolon was missing.

See Also

Using Templates (see page 462)

Function Templates (see page 433)

Class Templates (see page 412)

Template Compiler Switches (see page 414)

Template Generation Semantics (see page 414)

Exporting And Importing Templates (see page 432)

3.1.3.1.18.2 Using Templates

Templates, also called generics or parameterized types, let you construct a family of related functions or classes. These topics introduce the basic concept of templates:

Exporting and importing templates (see page 432)

Template Body Parsing (see page 461)

Function Templates (see page 433)

Class Templates (see page 412)

Implicit and Explicit Template Functions (see page 434)

Template Compiler Switches (see page 414)

The Keyword template (see page 568)

3.1.3.1.19 C++0x Features (C++Builder 2009)

C++Builder 2009 implements a number of the new features proposed in the C++0x standard. This section lists and describes these C++0x features. You can view the Working Paper draft that was used to guide the implementation of these features at Draft Working Paper. You can view a summary of the working group papers at Summary of Working Group Papers.

Topics

Name	Description
alignof Operator (C++0x) (see page 463)	The C++0x standard includes the **alignof** keyword and operator, which tells you the alignment of a type. To get the alignment of a type, use the following syntax:
Type Trait Functions (C++0x) (see page 463)	This section contains Type Trait Functions help topics.
Attributes noreturn and final (C++0x) (see page 492)	The C++0x standard includes the addition of attributes that might be applied to the declaration of a class, a general function, a constructor, an array, and so forth. C++Builder 2009 implements two attributes: **noreturn** and **final**. Attributes are set off in the code by double brackets, such as `[[noreturn]]`.
Explicit Conversion Operators (C++0x) (see page 493)	C++Builder 2009 includes support for explicit conversion operators, one of the features in the C++0x standard. You can now apply the function specifier **explicit** in the definition of a user-defined conversion operator. Previously, explicit constructors (including copy constructors) were added to the language in order to prevent unintended conversions being implicitly called by the compiler. Now explicit conversion operators have been added to provide the same control over unintended conversion calls. . Conversion functions declared as explicit work in the same contexts as explicit constructors (that is, direct-initialization, explicit type conversion). Explicit conversion operators produce compiler diagnostics in the same... more (see page 493)
extern Templates (C++0x) (see page 494)	C++Builder 2009 includes the use of **extern** templates, which allow you to define templates that are not instantiated in a translation unit. Using **extern** templates thus reduces both compilation time and the size of the compiled module. This feature is part of the new C++0x standard.
Forward Declaration of Enums (C++0x) (see page 495)	C++Builder 2009 introduces forward declaration of enums. You can declare an enumeration without providing a list of enumerators. Such declarations would not be definitions and can be provided only for enumerations with fixed underlying types. An enumeration can then be re-declared, possibly providing the missing list of enumerators, but the re-declaration must match the previous declaration. This feature is one of the C++0x features added to C++Builder 2009.
rvalue References (C++0x) (see page 495)	C++Builder 2009 includes the use of rvalue references, which allow creating a reference to temporaries. Also, rvalue references avoid unnecessary copying and make possible perfect forwarding functions. This feature is one of the C++0x features.

Static Assertions (C++0x) (see page 498)	The **static_assert** keyword is used to test assertions at compile time. This is one of the C++0x features added to C++Builder 2009. This keyword operates differently than the macro **assert**, which raises assertions at run time. The keyword **static_assert** also differs from the preprocessor directive **#error**, which operates during preprocessing and simply emits a message.
Strongly Typed Enums (C++0x) (see page 498)	C++Builder 2009 introduces scoped enums. In addition, existing enums are extended with underlying type and explicit scoping. This feature is one of the C++0x features added to C++Builder 2009. Scoped enums are generally characterized as follows: • Enumerators are in the scope of their enum. • Enumerators and enums do not implicitly convert to int (as do "plain" enumerators and enums). • Enums and their enumerators can have a defined underlying type.
Type Specifier decltype (C++0x) (see page 499)	The C++0x standard includes the **decltype** keyword and operator, which represents the type of an expression. This feature is one of the C++0x features added to C++Builder 2009.
Unicode Character Types and Literals (C++0x) (see page 500)	C++Builder 2009 implements new character types and character literals for Unicode. These types are among the C++0x features added to C++Builder 2009.

3.1.3.1.19.1 alignof Operator (C++0x)

The C++0x standard includes the **alignof** keyword and operator, which tells you the alignment of a type.

To get the alignment of a type, use the following syntax:

```
alignof(type);
```

The result is an integer constant of type `std::size_t`. The value indicates the boundaries on which elements of that type are aligned in memory. For instance, an alignment of 2 means that the type must begin on even memory addresses. A typical value for `alignof (double)` might be 8.

Applying **alignof** to a reference type yields the alignment of the referenced type. If you apply **alignof** to an array type, you get the alignment of its element's type.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Working Draft - Standard for Programming Language C++ (Sec. 5.3.6)

3.1.3.1.19.2 Type Trait Functions (C++0x)

This section contains Type Trait Functions help topics.

Topics

Name	Description
Type Trait Functions Overview (C++0x) (see page 472)	C++Builder 2009 supports a library of type trait functions designed to support compile time metaprogramming techniques. These type trait functions are intrinsic type functions that are defined in a manner similar to **typeid**, **sizeof**, and **decltype**. The type trait functions accept a type at compile time and deliver a compile time constant expression as a result, typically of type **bool**. Each type trait function is named after its respective type trait, prefixed with a double underscore (__), which marks a name reserved to the implementation. For example, the following is a type trait function that evaluates... more (see page 472)

__alignment_of (see page 473)	**Category** Type Trait Functions **Syntax:** `unsigned int __alignment_of( typename T )` This function is not necessary, as alignof (see page 531) is a proposed new keyword.
__array_extent (see page 473)	**Category** Type Trait Functions **Syntax** `unsigned int __array_extent( typename T, unsigned intI )` Returns: If T is not an array type, or if it has rank less than I, or if `I` is 0 and T has type “array of unknown bound of U”, then 0; otherwise, the size of the `I`'th dimension of T.
__array_rank (see page 473)	**Category** Type Trait Functions **Syntax** `unsigned int __array_rank( typename T )` Returns: If T names an array type, an integer representing the number of dimensions of T; otherwise, 0..
__has_nothrow_assign (see page 473)	**Category** Type Trait Functions **Syntax** `bool __has_nothrow_assign ( typename T )` Returns true if and only if compiler can prove T has a copy assignment operator that it cannot throw. True if __has_trivial_assign(T). Also true if copy assignment operator has an empty exception specification. **Ox interaction** false if copy assignment is defined as deleted.
__has_nothrow_copy_constructor (see page 474)	**Category** Type Trait Functions **Syntax** `bool __has_nothrow_copy_constructor ( typename T )` Returns true if and only if compiler can prove T has a copy constructor that cannot throw. **Error** if T is an incomplete type. True if `__has_trivial_copy_constructor(T)`. **Ox interaction** false if copy constructor is defined as deleted.
__has_nothrow_default_constructor (see page 474)	**Category** Type Trait Functions **Syntax** `bool __has_nothrow_default_constructor (typename T )` Returns true if and only if T can prove T has a default constructor it cannot throw. **Error** if T is an incomplete type. True if `__has_trivial_default_constructor(T)`. True if default constructor has an empty exception specification. False (but well-formed) if a class type does not have a default constructor. False if T is a reference type. **Ox interaction** false if the default constructor is defined as deleted.

__has_trivial_assign (see page 475)	**Category** Type Trait Functions **Syntax** `bool __has_trivial_assign (typename T )` Returns true if and only if T has a trivial copy assignment operator. **Error** if T is an incomplete type. The definition from Section 20.4.4.3 of the Working Draft notes says a type T has a trivial copy assignment operator if T is neither const nor a reference type and T is one of: • a scalar type (or array thereof) • a class type with a trivial copy assignment operator According to Section 21.8 p11 of the Working Draft, a copy assignment operator for class X is trivial if: • It is... more (see page 475)
__has_trivial_copy_constructor (see page 475)	**Category** Type Trait Functions **Syntax** `bool __has_trivial_copy_constructor ( typename T )` Returns true if and only if T has a trivial default constructor. **Error** if T is an incomplete type. False (but well-formed) if a class type does not have a default constructor The definition (from Section 20.4.4.3 of the Working Draft) notes has a type T has a trivial copy constructor if it is in the list:: • a scalartype (or array thereof) • a reference type (or array thereof) • an array of class type with trivial destructor • a class type with a trivial copy constructor (12.8) According to Section 12.8... more (see page 475)
__has_trivial_default_constructor (see page 476)	**Category** Type Trait Functions **Syntax** `bool __has_trivial_default_constructor (typename T )` Returns true if and only if T has a trivial default constructor. **Error**if T is an incomplete type. False (but well-formed) if a class type does not have a default constructor. According to the definition from Section 20.4.4.3 of the Working Draft notes, a type T has a trivial copy assignment operator if: • a scalar type (or array thereof) • an array of class type with a trivial default constructor • a class type with a trivial default constructorr Especially note false for reference types. According to Section 21.1 p5 of... more (see page 476)

__has_trivial_destructor (see page 477)	**Category** Type Trait Functions **Syntax** `bool __has_trivial_destructor (typename T )` Returns true if and only if T has a trivial destructor. **Error**if T is an incomplete type. Note: Definition from 20.4.4.3 notes has a type T has a trivial destructor if T is one of: • a scalartype (or array thereof) • a scalartype (or array thereof) • an array of class type with a trivial destructor • an array of class type with a trivial destructor • a class type with a trivial destructor According to Section 12.4, p 3 of the Working Draft, a destructor for class X is trivial if: • it... more (see page 477)
__has_virtual_destructor (see page 478)	**Category** Type Trait Functions **Syntax** `bool __has_virtual_destructor (typename T )` Returns true if and only if T is a class type and the destructor is declared virtual. **Error** if T is an incomplete type. Derived classes have a virtual destructor if the a base class destructor is declared virtual, even if not explicitly declared as virtual in the derived class.
__is_abstract (see page 478)	**Category** Type Trait Functions **Syntax** `bool __is_abstract( typename T )` Returns true if and only if T is an abstract class type. **Error** if T is an incomplete type. **Note:** An abstract class is one that contains or inherits (without overriding) at least one pure virtual function, according to Section 10.4 of the Working Draft.
__is_arithmetic (see page 478)	**Category** Type Trait Functions **Syntax** `bool __is_arithmetic ( typename T )` Returns __is_integral(T) || __is_floating_point(T).
__is_array (see page 479)	**Category** Type Trait Functions **Syntax** `bool __is_array( typename T )` Returns true if and only if T is an array type. **Note:** False for decayed pointer-to-array and reference to array; true for array-of-unknown bounds.
__is_base_of (see page 479)	**Category** Type Trait Functions **Syntax** `bool __is_base_of (typename Base, typename Derived)` Returns true if and only if Base is a base class of Derived. **Error** if Derived is an incomplete type and Base is a class-type.. Note: False (but well formed) if either Base or Derived is a union or non-class type, even if the other is an incomplete type. True if any class in the DAG of base classes for Derived is Base. This includes private, protected, ambiguous or virtual bases so simply return true the first time the search finds a match.

__is_class (see page 479)	**Category** Type Trait Functions **Syntax** `bool __is_class(typename T)` Returns true if and only if T is a class type, and NOT a union type. Returns true for classes declared with the class key class or struct. Returns false for reference/pointer to class type. Returns true for specialization of a class template. Ill-formed if called with the name of a template, without specifying the template parameters. A template is not a type; it is a type generator.
__is_complete_type (typename T) (see page 480)	**Category** Type Trait Functions **Syntax** `bool __is_complete_type(T)` Returns True if and only if T is a complete type at the point of the function call. This is a support function to help users track ill-formed code, not a distinct type trait required by the standard. This function is typically used in `static_assert` statements, because most other uses risk violating the ODR. Note that void and arrays of unknown bound are always incomplete types.
__is_compound (see page 480)	**Category** Type Trait Functions **Syntax** `bool __is_compound( typename T )` Returns true if and only if T is a compound type.. A compound type is essentially one item from the following list: • array • function • pointer • pointer-to-member • reference • class • union • enumeration Compound types are defined in Section 3.92 of the Working Draft.
__is_const (see page 481)	**Category** Type Trait Functions **Syntax** `bool __is_const(typename T)` Returns true if and only if T is a const-qualified type. Note: References are never cv-qualified. For pointers, refers to the pointer type itself, and NOT the pointed-to type. Returns true if T is both const and volatile qualified.
__is_convertible (see page 481)	**Category** Type Trait Functions **Syntax** `bool __is_convertible ( typename From, typename To )` Returns true if and only if **From** is implicitly convertible to **To**. Error if either From or To is an incomplete type. Note: Exact test described in Working Draft Section 20.4.4.3 is true if the following code is well formed:

__is_empty (see page 481)	**Category** Type Trait Functions **Syntax** `bool __is_empty( typename T )` Returns true if and only if T is an 'empty' type. **Error** if T is an incomplete type Definition of `__is_empty` is given in the table in Section 20.4.4.3 of the Working Draft. A type T is *empty* if T: • is a class type but not a union type. • has no non-static data members other than bit-fields of length 0 • has no virtual member functions • has no virtual base classes • has no base class which is not *empty*.
__is_enum (see page 482)	**Category** Type Trait Functions **Usage:** `bool _is_enum ( typename T )` Returns true if and only if T is an enum type. Returns true for the C++0x strongly typed enums as well. Returns false for reference/pointer to enum type.
__is_floating_point (see page 482)	**Category** Type Trait Functions **Syntax** `bool __is_floating_point(typename T)` Returns true if and only if T is a (potentially cv-qualified) floating point type. The standard set of floating point types is: • floating • double • long double Floating point types are defined in Section 3.9.1 p 7 of the Working Draft.
__is_function (see page 483)	**Category** Type Trait Functions **Syntax** `bool __is_function( typename T )` Returns true if and only if T is a function type. Returns false for reference/pointer to function type. Returns true for specialization of a function template. Ill-formed if called with the name of a template, without specifying the template parameters. A template is not a type; it is a type generator.
__is_fundamental (see page 483)	**Category** Type Trait Functions **Syntax** `bool __is_fundamental( typename T )` Returns true if and only if T is a fundamental type. Fundamental types are defined in Section 3.9.1 of the Working Draft. Definition is essentially `__is_arithmetic(T) || __is_void(T)` Alternative is a compound type, such as pointer, reference, enum, class or array.

__is_integral (see page 484)	**Category** Type Trait Functions **Syntax** `bool __is_integral(typename T)` Returns True if and only if T is an (potentially cv-qualified) integral type. Integral types are defined in the Working Standard, Section 3.9.1 page 7. The standard set of integral types is: bool, char, signed char, unsigned char, char16_t, char32_t, wchar_t, [unsigned]short, [unsigned] int, [unsigned] long, and [unsigned] long.
__is_lvalue_expr (see page 484)	**Category** Type Trait Functions **Syntax** `bool __is_lvalue_expr( typename T )` Returns true if and only if T is an lvalue expression.
__is_lvalue_reference (see page 484)	**Category** Type Trait Functions **Syntax** `bool __is_lvalue_reference( typename T )` Returns true if and only if T is an lvalue reference type. Can be a reference to an object or function type.
__is_member_function_pointer (see page 485)	**Category** Type Trait Functions **Syntax** `bool __is_member_function_pointer( typename T )` Returns true if and only if T is a pointer-to-member-function type . Returns false for a pointer-to-data member. Returns false for a regular object pointer.
__is_member_object_pointer (see page 485)	**Category** Type Trait Functions **Syntax** `bool __is_member_object_pointer( typename T )` Returns true if and only if T is a pointer-to-data-member type. Returns false for pointer-to-member-function. Returns false for a regular object pointer.
__is_member_pointer (see page 485)	**Category** Type Trait Functions **Syntax** `bool __is_member_pointer( typename T )` Returns: `__is_member_object_pointer(T) || __is_member_function_pointer(T)`.
__is_object (see page 485)	**Category** Type Trait Functions **Syntax** `bool __is_object( typename T)` Returns true if and only if T is an object type. Defined in Section 3.9 p8 of the Working Draft, essentially: `!__is_reference(T) && !__is_function(T) && !__is_void(T)`.
__is_pod (see page 486)	**Category** Type Trait Functions **Syntax** `bool __is_pod( typename T )` Returns true if and only if T is a POD (plain ol' data) type. **Error** if T is an incomplete type. POD types are defined in Section 3.9 p10 of the Working Draft. POD classes are defined in Section 9 p9 of the Working Draft. Effectively __is_trivial_type(T) && is standard_layout(T)
__is_pointer (see page 486)	**Category** Type Trait Functions **Syntax** `bool __is_pointer( typename T )` Returns true if and only if T is a pointer type. Can be an object pointer or a function pointer. False for pointer-to-member.

__is_polymorphic (see page 487)	**Category** Type Trait Functions **Syntax** `bool __is_polymorphic( typename T )` Returns true if and only if T is a poiymorphic class type. **Error** if T is an incomplete type. Polymorphic classes are defined in Section 10.3 of the Working Draft. "A class that declares or inherits a virtual function is called a polymorphic class."
__is_reference (see page 487)	**Category** Type Trait Functions **Syntax** `bool __is_reference( typename T )` Returns: `__is_lvalue_reference(T) || __is_rvalue_reference(T)`.
__is_rvalue_expr (see page 487)	**Category** Type Trait Functions **Syntax** `bool __is_rvalue_expr( typename T )` Returns true if and only if T is an rvalue expression.
__is_rvalue_reference (see page 488)	**Category** Type Trait Functions **Usage:** `bool __is_rvalue_reference( typename T )` Returns true if and only if T is an rvalue reference type. Can be a reference to an object or function type, even though function type rvalues make little sense beyond metaprogramming in the type system.
__is_same (see page 488)	**Category** Type Trait Functions **Syntax** `bool __is_same( typename T, typename U )` Returns true if and only if T and U are the same type, with identical CV qualifiers. Names of types when calling `__is_same` may be different aliases for the same type, if called with typedefs or template type parameters. **Ox** interaction: template aliases will allow another way to alias the same type name.

__is_scalar (see page 488)	**Category**
	Type Trait Functions
	Syntax
	`bool __is_scalar( typename T )`
	Returns true if and only if T is a scalar type.
	Scalar type is defined in Section 3.9 p10 of the Working Draft, essentially:
	`__is_arithmetic( T ) || __is_enumeration(T) || __is_pointer(T) || __is_member_pointer(T)`
	Scalar types have a built in meaning for operator < as well as operator ==. Therefore, `__closure` is not a scalar type.

__is_signed (see page 489)	**Category** Type Trait Functions **Syntax** `_bool __is_signed( typename T )` Returns true if and only if `__is_arithmetic(T)` and T recognizes values less than zero.

__is_standard_layout (see page 489)	**Category** Type Trait Functions **Syntax** `bool __is_standard_layout(typename T)` Returns true if and only if T is a standard layout type. **Error** if T is an incomplete type. Standard layout classes are defined in Section 9, p 6 of the Working Draft. A standard layout type is: • a scalar type • an array of standard layout types • a standard layout class type: A standard layout class is a class that has: • no non-static data members other than standard layout types • same access control for all non-static data members (such as all public or private) • no base classes other than standard layout classes... more (see page 489)
__is_trivial (see page 490)	**Category** Type Trait Functions **Syntax** `bool __is_trivial( typename T )` Returns true if and only if T is a trivial type. **Error** if T is an incomplete type. Trivial types are defined in Section 3.9 p10 of the Working Draft. Trivial classes are defined in Section 9 p5 of the Working Draft. Returns true if T is both const and volatile qualified. A type is trivial if it is: • a scalar type • an array of trivial types • a class/union type where all 4 special members are trivial: • default constructor • copy constructor • copy assignment operator • destructor Consult this online help for... more (see page 490)
__is_union (see page 491)	**Category** Type Trait Functions **Syntax** `bool __is_union( typename T )` Returns true if and only if T is a union type. Note: unions are class types declared with the class-key union. Returns false for reference/pointer to union type.
__is_unsigned (typename T) (see page 491)	**Category** Type Trait Functions **Syntax** `bool __is_unsigned` Returns true if `_is_arithmetic(T)` and T does not recognize values less than zero

__is_void (see page 491)	**Category** Type Trait Functions **Syntax** `bool __is_void( typename T )` Returns True if T is (potentially cv-qualified) void. Returns False otherwise, including for pointer or reference to void.
__is_volatile (see page 491)	**Category** Type Trait Functions **Syntax** `bool __is_volatile( typename T )` Returns true if and only if T is a volatile-qualified type.. Note: References are never cv-qualified. For pointers, refers to the pointer type itself, and NOT the pointed-to type. Returns true if T is both const and volatile qualified.

3.1.3.1.19.2.1 Type Trait Functions Overview (C++0x)

C++Builder 2009 supports a library of type trait functions designed to support compile time metaprogramming techniques.

These type trait functions are intrinsic type functions that are defined in a manner similar to **typeid**, **sizeof**, and **decltype**. The type trait functions accept a type at compile time and deliver a compile time constant expression as a result, typically of type **bool**.

Each type trait function is named after its respective type trait, prefixed with a double underscore (__), which marks a name reserved to the implementation.

For example, the following is a type trait function that evaluates True if T is a union type, or False otherwise:

```
bool __is_union(typename T)
```

The **typename** keyword here indicates a function that takes a type rather than a value as an argument, for illustration purposes only.

Differences Between Type Trait Functions and Other Intrinsic Functions

Type trait functions accept only named types, not arbitrary expressions (which would be accepted by **typeid**, **sizeof**, and **decltype**). You can specify multiple arguments with a type trait function, but a comma-separated list is interpreted as an application of the comma operator by **typeid**, **sizeof**, and **decltype**.

Similarly, any attempt to reference the type of a bit field (such as through **decltype**) produces the underlying storage type for the bit field, and that is what is tested in the type trait functions.

Incomplete Types and Type Trait Functions

Many of the type trait functions do work with incomplete types. For example, after a class has been forward declared, the type can be identified as a class, not a union, enum or fundamental type. The class can have references and pointers, and so forth. Therefore, most type trait functions accept two special incomplete types: void and array-of-unknown-bound (of complete types). For these two incomplete types, most type trait functions return False.

However, a few type functions require complete types. For example, a class declaration is not sufficient to know about bases or triviality.

For convenience, the following type trait function is implemented in C++Builder:

```
__is_complete_type(T)
```

This type trait function is unique in that it might return different results at compile time depending on its location in the translation unit. This phenomenon makes this type trait function ripe for ODR violations when used carelessly.

A table of type trait functions that require that the type be a complete type, an array of unknown bound, or (possibly cv-qualified) `void` is given in **Table 43**: “Type Property Predicates” in Section 20 of the Working Draft.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Working Draft — Standard for Programming Language C++: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.2 __alignment_of

Category

Type Trait Functions

Syntax:

```
unsigned int __alignment_of( typename T )
```

This function is not necessary, as alignof (see page 531) is a proposed new keyword.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 463)

3.1.3.1.19.2.3 __array_extent

Category

Type Trait Functions

Syntax

```
unsigned int __array_extent( typename T, unsigned intI )
```

Returns: If T is not an array type, or if it has rank less than I, or if `I` is 0 and `T` has type “array of unknown bound of U”, then 0; otherwise, the size of the `I`’th dimension of `T`.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.4 __array_rank

Category

Type Trait Functions

Syntax

```
unsigned int __array_rank( typename T )
```

Returns: If T names an array type, an integer representing the number of dimensions of T; otherwise, 0..

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.5 __has_nothrow_assign

Category

Type Trait Functions

Syntax

```
bool __has_nothrow_assign ( typename T )
```

Returns true if and only if compiler can prove T has a copy assignment operator that it cannot throw.

True if __has_trivial_assign(T).

Also true if copy assignment operator has an empty exception specification.

Ox interaction false if copy assignment is defined as deleted.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.6 __has_nothrow_copy_constructor

Category

Type Trait Functions

Syntax

```
bool __has_nothrow_copy_constructor ( typename T )
```

Returns true if and only if compiler can prove T has a copy constructor that cannot throw.

Error if T is an incomplete type.

True if `__has_trivial_copy_constructor(T)`.

Ox interaction false if copy constructor is defined as deleted.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.7 __has_nothrow_default_constructor

Category

Type Trait Functions

Syntax

```
bool __has_nothrow_default_constructor (typename T )
```

Returns true if and only if T can prove T has a default constructor it cannot throw.

Error if T is an incomplete type.

True if `__has_trivial_default_constructor(T)`.

True if default constructor has an empty exception specification.

False (but well-formed) if a class type does not have a default constructor.

False if T is a reference type.

Ox interaction false if the default constructor is defined as deleted.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.8 __has_trivial_assign

Category

Type Trait Functions

Syntax

```
bool __has_trivial_assign (typename T )
```

Returns true if and only if T has a trivial copy assignment operator.

Error if T is an incomplete type.

The definition from Section 20.4.4.3 of the Working Draft notes says a type T has a trivial copy assignment operator if T is neither const nor a reference type and T is one of:

- a scalar type (or array thereof)
- a class type with a trivial copy assignment operator

According to Section 21.8 p11 of the Working Draft, a copy assignment operator for class X is trivial if:

- It is not user provided
- class X has no virtual functions
- class X has no virtual base classes
- each direct base class of X has a trivial copy assignment operator.
- for all the non-static data members of X that are of class type (or array thereof), each such class type has a trivial copy assignment operator;

A copy assignment operator is not user provided if it is implicitly declared, or defined inline as `= default;`.

Ox interaction false if the copy assign operator is defined as deleted.

Ox interaction false if the copy assign operator is defined as deleted.

Ox interaction false if the default constructor is defined as deleted.

Ox interaction with default function definitions.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft - Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.9 __has_trivial_copy_constructor

Category

Type Trait Functions

Syntax

```
bool __has_trivial_copy_constructor ( typename T )
```

Returns true if and only if T has a trivial default constructor.

Error if T is an incomplete type.

False (but well-formed) if a class type does not have a default constructor

The definition (from Section 20.4.4.3 of the Working Draft) notes has a type T has a trivial copy constructor if it is in the list::

- a scalartype (or array thereof)
- a reference type (or array thereof)
- an array of class type with trivial destructor
- a class type with a trivial copy constructor (12.8)

According to Section 12.8 p6 of the Working Draft:

A copy constructor for class X is trivial if:

- It is not user provided (see Section 8.4)
- class X has no virtual functions
- class X has no virtual base classes
- each direct base class of X has a trivial copy constructor.
- for all the non-static data members of X that are of class type (or array thereof), each such class type has a trivial copy constructor;.

A copy constructor is not user provided if it is implicitly declared, or defined inline as `= default;`

Ox interaction false if the default constructor is defined as deleted.

Ox interactionwith default function definitions.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft - Standard for Programming Language C++: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.10 __has_trivial_default_constructor

Category

Type Trait Functions

Syntax

```
bool __has_trivial_default_constructor (typename T )
```

Returns true if and only if T has a trivial default constructor.

Errorif T is an incomplete type.

False (but well-formed) if a class type does not have a default constructor.

According to the definition from Section 20.4.4.3 of the Working Draft notes, a type T has a trivial copy assignment operator if:

- a scalar type (or array thereof)
- an array of class type with a trivial default constructor
- a class type with a trivial default constructorr

Especially note false for reference types.

According to Section 21.1 p5 of the Working Draft, a default constructor is trivial if:

- It is not user provided

- Its class has no virtual functions
- Its class has no virtual base classes
- Each direct base class of X has a trivial copy assignment operator.
- All the direct base classes of its class have trivial default constructors.
- For all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

A default consructor is not user provided if it is implicitly declared, or defined inline as `= default;`.

Ox interaction if the copy assign operator is defined as deleted.

Ox interaction false if the default constructor is defined as deleted.

Ox interaction with default function definitions.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft - Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.11 __has_trivial_destructor

Category

Type Trait Functions

Syntax

```
bool __has_trivial_destructor (typename T )
```

Returns true if and only if T has a trivial destructor.

Errorif T is an incomplete type.

Note: Definition from 20.4.4.3 notes has a type T has a trivial destructor if T is one of:

- a scalartype (or array thereof)
- a scalartype (or array thereof)
- an array of class type with a trivial destructor
- an array of class type with a trivial destructor
- a class type with a trivial destructor

According to Section 12.4, p 3 of the Working Draft, a destructor for class X is trivial if:

- it is not user-provided
- class X has no virtual functions
- class X has no virtual base classes
- all of the direct base classes of its class have trivial destructors
- for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

A destructor is not user provided if it is implicitly declared, or defined inline as = default;

Ox interaction false if the destructor is defined as deleted.

Ox interaction with default function definitions.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.12 __has_virtual_destructor

Category

Type Trait Functions

Syntax

```
bool __has_virtual_destructor (typename T )
```

Returns true if and only if T is a class type and the destructor is declared virtual.

Error if T is an incomplete type.

Derived classes have a virtual destructor if the a base class destructor is declared virtual, even if not explicitly declared as virtual in the derived class.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.13 __is_abstract

Category

Type Trait Functions

Syntax

```
bool __is_abstract( typename T )
```

Returns true if and only if T is an abstract class type.

Error if T is an incomplete type.

Note: An abstract class is one that contains or inherits (without overriding) at least one pure virtual function, according to Section 10.4 of the Working Draft.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.14 __is_arithmetic

Category

Type Trait Functions

Syntax

```
bool __is_arithmetic ( typename T )
```

Returns __is_integral(T) || __is_floating_point(T).

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.15 __is_array

Category

Type Trait Functions

Syntax

```
bool __is_array( typename T )
```

Returns true if and only if T is an array type.

Note: False for decayed pointer-to-array and reference to array; true for array-of-unknown bounds.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.16 __is_base_of

Category

Type Trait Functions

Syntax

```
bool __is_base_of (typename Base, typename Derived)
```

Returns true if and only if Base is a base class of Derived.

Error if Derived is an incomplete type and Base is a class-type..

Note: False (but well formed) if either Base or Derived is a union or non-class type, even if the other is an incomplete type.

True if any class in the DAG of base classes for Derived is Base. This includes private, protected, ambiguous or virtual bases so simply return true the first time the search finds a match.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.17 __is_class

Category

Type Trait Functions

Syntax

```
bool __is_class(typename T)
```

Returns true if and only if T is a class type, and NOT a union type.

Returns true for classes declared with the class key class or struct.

Returns false for reference/pointer to class type.

Returns true for specialization of a class template.

Ill-formed if called with the name of a template, without specifying the template parameters. A template is not a type; it is a type generator.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.18 __is_complete_type (typename T)

Category

Type Trait Functions

Syntax

```
bool __is_complete_type(T)
```

Returns True if and only if T is a complete type at the point of the function call.

This is a support function to help users track ill-formed code, not a distinct type trait required by the standard. This function is typically used in `static_assert` statements, because most other uses risk violating the ODR. Note that void and arrays of unknown bound are always incomplete types.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.19 __is_compound

Category

Type Trait Functions

Syntax

```
bool __is_compound( typename T )
```

Returns true if and only if T is a compound type..

A compound type is essentially one item from the following list:

- array
- function
- pointer
- pointer-to-member
- reference
- class
- union
- enumeration

Compound types are defined in Section 3.92 of the Working Draft.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft - Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.20 __is_const

Category

Type Trait Functions

Syntax

```
bool __is_const(typename T)
```

Returns true if and only if T is a const-qualified type.

Note: References are never cv-qualified.

For pointers, refers to the pointer type itself, and NOT the pointed-to type.

Returns true if T is both const and volatile qualified.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.21 __is_convertible

Category

Type Trait Functions

Syntax

```
bool __is_convertible ( typename From, typename To )
```

Returns true if and only if **From** is implicitly convertible to **To**.

Error if either From or To is an incomplete type.

Note: Exact test described in Working Draft Section 20.4.4.3 is true if the following code is well formed:

```
template <class T>
typename add_rvalue_reference<T>::type create();

To test() { return create<From>(); |
```

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft - Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.22 __is_empty

Category

Type Trait Functions

Syntax

```
bool __is_empty( typename T )
```

Returns true if and only if T is an 'empty' type.

Error if T is an incomplete type

Definition of `__is_empty` is given in the table in Section 20.4.4.3 of the Working Draft.

A type T is *empty* if T:

- is a class type but not a union type.
- has no non-static data members other than bit-fields of length 0
- has no virtual member functions
- has no virtual base classes
- has no base class which is not *empty*.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.23 __is_enum

Category

Type Trait Functions

Usage: `bool _is_enum ( typename T )`

Returns true if and only if T is an enum type.

Returns true for the C++0x strongly typed enums as well.

Returns false for reference/pointer to enum type.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.24 __is_floating_point

Category

Type Trait Functions

Syntax

```
bool __is_floating_point(typename T)
```

Returns true if and only if T is a (potentially cv-qualified) floating point type.

The standard set of floating point types is:

- floating
- double

- long double

Floating point types are defined in Section 3.9.1 p 7 of the Working Draft.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.25 __is_function

Category

Type Trait Functions

Syntax

```
bool __is_function( typename T )
```

Returns true if and only if T is a function type.

Returns false for reference/pointer to function type.

Returns true for specialization of a function template.

Ill-formed if called with the name of a template, without specifying the template parameters. A template is not a type; it is a type generator.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions (C++0x) (see page 472)

3.1.3.1.19.2.26 __is_fundamental

Category

Type Trait Functions

Syntax

```
bool __is_fundamental( typename T )
```

Returns true if and only if T is a fundamental type.

Fundamental types are defined in Section 3.9.1 of the Working Draft.

Definition is essentially `__is_arithmetic(T) || __is_void(T)`

Alternative is a compound type, such as pointer, reference, enum, class or array.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.27 __is_integral

Category

Type Trait Functions

Syntax

```
bool __is_integral(typename T)
```

Returns True if and only if T is an (potentially cv-qualified) integral type.

Integral types are defined in the Working Standard, Section 3.9.1 page 7.

The standard set of integral types is: bool, char, signed char, unsigned char, char16_t, char32_t, wchar_t, [unsigned]short, [unsigned] int, [unsigned] long, and [unsigned] long.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.28 __is_lvalue_expr

Category

Type Trait Functions

Syntax

```
bool __is_lvalue_expr( typename T )
```

Returns true if and only if T is an lvalue expression.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.29 __is_lvalue_reference

Category

Type Trait Functions

Syntax

```
bool __is_lvalue_reference( typename T )
```

Returns true if and only if T is an lvalue reference type.

Can be a reference to an object or function type.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.30 __is_member_function_pointer

Category

Type Trait Functions

Syntax

```
bool __is_member_function_pointer( typename T )
```

Returns true if and only if T is a pointer-to-member-function type .

Returns false for a pointer-to-data member.

Returns false for a regular object pointer.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.31 __is_member_object_pointer

Category

Type Trait Functions

Syntax

```
bool __is_member_object_pointer( typename T )
```

Returns true if and only if T is a pointer-to-data-member type.

Returns false for pointer-to-member-function.

Returns false for a regular object pointer.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.32 __is_member_pointer

Category

Type Trait Functions

Syntax

```
bool __is_member_pointer( typename T )
```

Returns: `__is_member_object_pointer(T) || __is_member_function_pointer(T)`.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.33 __is_object

Category

Type Trait Functions

Syntax

```
bool __is_object( typename T)
```

Returns true if and only if T is an object type.

Defined in Section 3.9 p8 of the Working Draft, essentially:

`!__is_reference(T) && !__is_function(T) && !__is_void(T)`.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.34 __is_pod

Category

Type Trait Functions

Syntax

```
bool __is_pod( typename T )
```

Returns true if and only if T is a POD (plain ol' data) type.

Error if T is an incomplete type.

POD types are defined in Section 3.9 p10 of the Working Draft.

POD classes are defined in Section 9 p9 of the Working Draft.

Effectively __is_trivial_type(T) && is standard_layout(T)

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - ttp://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.35 __is_pointer

Category

Type Trait Functions

Syntax

```
bool __is_pointer( typename T )
```

Returns true if and only if T is a pointer type.

Can be an object pointer or a function pointer.

False for pointer-to-member.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.36 __is_polymorphic

Category

Type Trait Functions

Syntax

```
bool __is_polymorphic( typename T )
```

Returns true if and only if T is a poiymorphic class type.

Error if T is an incomplete type.

Polymorphic classes are defined in Section 10.3 of the Working Draft.

"A class that declares or inherits a virtual function is called a polymorphic class."

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.37 __is_reference

Category

Type Trait Functions

Syntax

```
bool __is_reference( typename T )
```

Returns: `__is_lvalue_reference(T) || __is_rvalue_reference(T)`.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.38 __is_rvalue_expr

Category

Type Trait Functions

Syntax

```
bool __is_rvalue_expr( typename T )
```

Returns true if and only if T is an rvalue expression.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.39 __is_rvalue_reference

Category

Type Trait Functions

Usage: `bool __is_rvalue_reference( typename T )`

Returns true if and only if T is an rvalue reference type.

Can be a reference to an object or function type, even though function type rvalues make little sense beyond metaprogramming in the type system.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.40 __is_same

Category

Type Trait Functions

Syntax

```
bool __is_same( typename T, typename U )
```

Returns true if and only if T and U are the same type, with identical CV qualifiers.

Names of types when calling `__is_same` may be different aliases for the same type, if called with typedefs or template type parameters.

Ox interaction: template aliases will allow another way to alias the same type name.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.41 __is_scalar

Category

Type Trait Functions

Syntax

```
bool __is_scalar( typename T )
```

Returns true if and only if T is a scalar type.

Scalar type is defined in Section 3.9 p10 of the Working Draft, essentially:

```
__is_arithmetic( T ) || __is_enumeration(T) ||
__is_pointer(T) || __is_member_pointer(T)
```

Scalar types have a built in meaning for operator < as well as operator ==. Therefore, `__closure` is not a scalar type.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.42 __is_signed

Category

Type Trait Functions

Syntax

```
_bool __is_signed( typename T )
```

Returns true if and only if `__is_arithmetic(T)` and T recognizes values less than zero.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.43 __is_standard_layout

Category

Type Trait Functions

Syntax

```
bool __is_standard_layout(typename T)
```

Returns true if and only if T is a standard layout type.

Error if T is an incomplete type.

Standard layout classes are defined in Section 9, p 6 of the Working Draft.

A standard layout type is:

- a scalar type
- an array of standard layout types
- a standard layout class type:

A standard layout class is a class that has:

- no non-static data members other than standard layout types
- same access control for all non-static data members (such as all public or private)
- no base classes other than standard layout classes
- no virtual functions and no virtual base classes
- no more than one class in the DAG of inheritance tree has any non-static data
- no base class of same type as the first (if any) non-static data member

Standard layout types trigger several special clauses in the standard, which may affect code generation and optimizations, or simply whether certain constructs are undefined. In particular, standard layout classes require the empty base optimization.

Note that standard layout types are a generalization of PODs in the 2003 standard, if the POD/standard layout refinement has

not happened when this type function is initially implanted.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

3.1.3.1.19.2.44 __is_trivial

Category

Type Trait Functions

Syntax

```
bool __is_trivial( typename T )
```

Returns true if and only if T is a trivial type.

Error if T is an incomplete type.

Trivial types are defined in Section 3.9 p10 of the Working Draft.

Trivial classes are defined in Section 9 p5 of the Working Draft.

Returns true if T is both const and volatile qualified.

A type is trivial if it is:

- a scalar type
- an array of trivial types
- a class/union type where all 4 special members are trivial:
- default constructor
- copy constructor
- copy assignment operator
- destructor

Consult this online help for the definition of each `__has_trivial_*` traits.

Trivial types trigger several special clauses in the standard, which may affect code generation and optimizations, or simply whether certain constructs are undefined.

Note that trivial types are a generalization of PODs (plain old data objects) in the 2003 standard, if the POD/trivial refinement has not happened when this type function is initially implanted.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

Working Draft — Standard for Programming Language C++ - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2588.pdf

__has_trivial_assign.xml (see page 475)

__has_trivial_copy_constructor (see page 475)

__has_trivial_copy_constructor.xml (see page 475)

__has_trivial_default_constructor.xml (see page 476)

3.1.3.1.19.2.45 __is_union

Category

Type Trait Functions

Syntax

```
bool __is_union( typename T )
```

Returns true if and only if T is a union type.

Note: unions are class types declared with the class-key union.

Returns false for reference/pointer to union type.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.46 __is_unsigned (typename T)

Category

Type Trait Functions

Syntax

```
bool __is_unsigned
```

Returns true if `_is_arithmetic(T)` and T does not recognize values less than zero

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.47 __is_void

Category

Type Trait Functions

Syntax

```
bool __is_void( typename T )
```

Returns True if T is (potentially cv-qualified) void. Returns False otherwise, including for pointer or reference to void.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.2.48 __is_volatile

Category

Type Trait Functions

Syntax

```
bool __is_volatile( typename T )
```

Returns true if and only if T is a volatile-qualified type..

Note: References are never cv-qualified.

For pointers, refers to the pointer type itself, and NOT the pointed-to type.

Returns true if T is both const and volatile qualified.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Type Trait Functions Overview (see page 472)

3.1.3.1.19.3 Attributes noreturn and final (C++0x)

The C++0x standard includes the addition of attributes that might be applied to the declaration of a class, a general function, a constructor, an array, and so forth. C++Builder 2009 implements two attributes: **noreturn** and **final**.

Attributes are set off in the code by double brackets, such as `[[noreturn]]`.

The noreturn Attribute

The **noreturn** attribute specifies that a function does not return. If a function marked **noreturn** is called and eventually executes a return statement, the program is considered ill-formed, but the compiler does not issue a message.

For example:

```
void f [[noreturn]] () {
   throw “error”;           // OK
}
void g [[noreturn]] (int i) {   // ill-formed if called with i<=0
   if (i > 0)
      throw “positive”;
}
```

The **noreturn** attribute is useful for a few library functions that cannot return, such as abort and exit. You can also define your own functions that never return by using the **noreturn** attribute.

For example:

```
void fatal(void) [[noreturn]];
void fatal(...)
{
...
        exit(1);
}
```

The noreturn keyword tells the C++ compiler to assume that fatal cannot return. The compiler can then optimize without regard to what would happen if fatal ever did return. Thus, using noreturn can make better code. More importantly, it helps avoid spurious warnings of uninitialized variables. You cannot assume that registers saved by the calling function are restored before calling the function with the noreturn attribute. It does not make sense for a noreturn function to have a return type other than void.

The final Attribute

The **final** attribute prevents a class or function from being further inherited. You can add the **final** attribute to a class definition or to a virtual member function declaration inside a class definition.

A class with the **final** attribute is not allowed to be a base class for another class. A virtual function with the **final** attribute is not overridden in a subclass. If the attribute is specified for a class definition, it is equivalent to being specified for each virtual

member function of that class, including inherited member functions.

If a virtual member function f in some class B is marked **final** and in a class D derived from B, a function D::f overrides B::f, the program is ill-formed (the compiler does not issue a message).

For example:

```
struct B {
   virtual void f [[ final ]] ();
};
struct D : B {
   void f();   // ill-formed
};
```

See Also

C++0x Features (C++Builder 2009) (see page 462)

noreturn (see page 557)

final (see page 550)

3.1.3.1.19.4 Explicit Conversion Operators (C++0x)

C++Builder 2009 includes support for explicit conversion operators, one of the features in the C++0x standard.

You can now apply the function specifier **explicit** in the definition of a user-defined conversion operator. Previously, explicit constructors (including copy constructors) were added to the language in order to prevent unintended conversions being implicitly called by the compiler. Now explicit conversion operators have been added to provide the same control over unintended conversion calls. .

Conversion functions declared as explicit work in the same contexts as explicit constructors (that is, direct-initialization, explicit type conversion). Explicit conversion operators produce compiler diagnostics in the same contexts (copy-initialization) as explicit constructors do.

For example:

```
class T { };

class X {

public:

explicit operator T() const;

};

void m() {

X x;


// with cast:

T tc = (T)x;// ok

// without cast:

T t = x;// error: E2034


// gets: Error E2034 x.cpp 13: Cannot convert 'X' to 'T' in function m()

}
```

See Also

C++0x Features (C++Builder 2009) (see page 462)

Draft Working Paper

Summary of Working Group Papers

3.1.3.1.19.5 extern Templates (C++0x)

C++Builder 2009 includes the use of **extern** templates, which allow you to define templates that are not instantiated in a translation unit. Using **extern** templates thus reduces both compilation time and the size of the compiled module. This feature is part of the new C++0x standard.

Explicit Instantiation and Declaration

An **extern** template allows you to declare a template without instantiating it in the translation unit.

To illustrate, the following both creates and instantiates a template:

```
template <class T>
class MyClass {
   //various code
}

template class MyClass<int>;
...
MyClass<int> myClass;
```

The line `template class MyClass<int>` is an explicit template definition and causes the template to be instantiated explicitly in its code unit, resulting in generating code for the template in that unit. Similarly, the line `MyClass<int> myClass;` implicitly instantiates the template, also resulting in code generation in the unit. If either of these lines of code are in your unit, the template is instantiated there.

However, suppose you want to have a library in which all instantiations of this template occur, and you want to refer to these instantiations in an executable. To make an explicit template declaration that does not instantiate the template in your code unit, use the following:

```
extern template class MyClass<int>;
```

You can then reference the template, but the compiler does not generate code for it in that translation unit.

extern Template Usage

Here are the rules for using extern templates:

- A template instantiation must either follow an explicit template declaration in that translation unit or be in another translation unit. This is similar to the usual use of **extern**: the entity referenced must be defined somewhere.
- An explicit template definition must occur only once in a unit.
- An explicit template definition must follow an explicit template declaration if both are present in a translation unit.
- An explicit template declaration can only apply to names of objects, functions, and explicit template instantiations. It may not refer to a static function, but may apply to a static member function.
- The **extern** specifier may not be used in declaration of class members or parameters.
- An explicit template declaration only suppresses code generation if that template has not been instantiated with the same specialization. For instance:

```
template class MyClass<int>;
...
extern template class MyClass<int>;    // not allowed
extern template class MyClass<float>;  // OK
```

See Also

C++0x Features (C++Builder 2009) (see page 462)

Working Draft - Standard for Programming Language C++ (Sec. 17.2)

3.1.3.1.19.6 Forward Declaration of Enums (C++0x)

C++Builder 2009 introduces forward declaration of enums. You can declare an enumeration without providing a list of enumerators. Such declarations would not be definitions and can be provided only for enumerations with fixed underlying types. An enumeration can then be re-declared, possibly providing the missing list of enumerators, but the re-declaration must match the previous declaration. This feature is one of the C++0x features added to C++Builder 2009.

```
enum E : short;           // OK: unscoped, underlying type is short
enum F:                   // illegal: enum-base is required
enum class G : short      // OK: scoped, underlying type is short
enum class H;             // OK: scoped, underlying type is int
enum E : short;           // OK: redeclaration of E
enum class G : short;     // OK: redeclaration of G
enum class H;             // OK: redeclaration of H
enum class H : int;       // OK: redeclaration of H
enum class E : short;     // illegal: previously declared as unscoped
enum G : short;           // illegal: previously declared as scoped
enum E;                   // illegal: enum-base is required
enum E : int              // illegal: different underlying type
enum class G;             // illegal: different underlying type
enum class H : short;     // illegal: different underlying type
enum class H {/* */}]     // OK:  this redeclaration is a definition
```

See Also

C++0x Features (C++Builder 2009) (see page 462)

Strongly Typed Enums (see page 498)

3.1.3.1.19.7 rvalue References (C++0x)

C++Builder 2009 includes the use of rvalue references, which allow creating a reference to temporaries. Also, rvalue references avoid unnecessary copying and make possible perfect forwarding functions. This feature is one of the C++0x features.

Description

Rvalue references are a compound type like standard C++ references, which are referred to as lvalue references. An lvalue reference is formed by appending the ampersand character (**&**) to a type:

```
SomeClass l;
SomeClass& lReference = l;    //lvalue rererence
```

The syntax of an rvalue reference is to add **&&** after a type:

```
SomeClass r;
SomeClass&& rReference = r;    //rvalue rererence
```

An rvalue reference behaves like an lvalue reference, except that you can bind an rvalue reference to a temporary -- an rvalue.

```
SomeClass a;
a = SomeClass();
SomeClass& lReference = a;          //OK-lvalue reference can bind to an lvalue such as "a"
SomeClass& lReference2 = SomeClass();   //error-lvalue reference can't bind to an rvalue
SomeClass&& rReference = SomeClass();   //OK for rvalue reference to bind to rvalue

// Both references can be used the same way
SomeOtherClass value1 = SomeFunc(lReference);
SomeOtherClass value2 = SomeFunc(rReference);
```

In the example above, `SomeClass()` is not bound to an identifiier, so it is an rvalue and can be bound to an rvalue reference -- but not an lvalue reference.

Eliminating Unnecessary Copying

In many cases, data is copied that simply needs to be moved, that is, the original data holder need not retain the data. An example is swapping data in two structures, so neither structure holds its previous data. It would be logically sufficient to simply switch the references to the data.

Rvalue rererences can be used to distinguish cases that require copying versus cases that merely require moving data. Since copying can be a lengthy operation, you want to avoid it if possible.

If a function wants to copy something that has been passed to it as an rvalue, then it can do a move rather than a copy, because it knows that the value is temporary. If a function is passed an lvalue, it may need to do a full copy if copy elision doesn't apply. You can distinguish these cases with the function signature.

Consider a class ExampleClass that has a clone function that makes a deep copy of class instances. You want to define a move function that moves an object's value. This function can be overloaded as follows:

```
// Parameter is lvalue
ExampleClass move(ExampleClass& l)
{
    return l.clone();   //returns a copy since we can't touch the original value

// Parameter is rvalue
ExampleClass move(ExampleClass&& r)
{
    return r;       //returns a reference since we don't care about the temporary's value
}
template class MyClass<int>;
...
extern template class MyClass<int>;    // not allowed
extern template class MyClass<float>;  // OK
```

We could then use the move function for both rvalues and lvalues:

```
ExampleClass a, b, c;
a = ExampleClass();
b = b.move(a);          //parameter is lvalue
c = c.move(ExampleClass()); //parameter is rvalue
```

Note that the move function for the rvalue parameter does very little, so it executes much more quickly than the move for an lvalue parameter.

You can use a similar technique for functions that need to make copies, such as copy constructors and assignment operators. Suppose we have a template class with a pointer to some other class, where clone is again a deep copy function:

```
template <class T>
class PointerClass
{
private:
    T* pointer;
public:
    // Regular constructor
    PointerClass(void);

    // Copy constructor for lvalues
    PointerClass(PointerClass& pcl) : pointer(pcl.pointer ? pcl.pointer.clone() : 0) {} //make
full copy
    // Copy constructor for rvalues
    PointerClass(PointerClass&& pcr) : pointer(pcr.pointer) {pcr.pointer = 0;}
```

The copy constructor that takes an rvalue:

- Does a move, not a copy. That is, it simply returns a reference to the data.

- Treats the rvalue argument **pcr** just like an lvalue in its code.
- Leaves the rvalue object in a defined state so that it can safely be deleted.

Non-Copyable but Movable Types

Types that are not copyable, such as ones that use unique_ptr, can be made movable. Although they cannot define assignment operators, they can implement move functions and swap functions, since these do not require copying when rvalue references are used. A sort function could be developed, since it only requires swapping, not copying.

For instance, consider this factory function that takes one argument:

```
template <class T, class U>
factory(const U& u)
{
    return new T(u);
}
```

The above definition of factory works in this case:

```
T* p = factory<T>(7);
```

However, a compiler error occurs when a T is used whose constructor's parameter is a non-const reference. You could fix this case by removing const from the definition:

```
template <class T, class U>
factory(U& u)
{
    return new T(u);
}
```

However, the previous example now fails:

```
T* p = factory<T>(7); // compiler error
T* u = new T(7);    //OK
```

This causes an error because the value 7 causes the template argument to be matched to int &, but this does not bind to the rvalue 7.

This could be remedied by defining a factory function for each case of const and non-const. This is problematic, because the number of functions needed increases exponentially with the number of arguments.

If you make the argument an rvalue reference, you can simplify the situation:

```
template <class T, class U.
factory(u&& u)
{
   return new T(forward<U>(u));
}
```

Now the argument **u** binds to both rvalues and lvalues. The `forward` function returns an rvalue or lvalue, exactly as it was passed. It can be defined in this way:

```
template <class U>
struct identity
{
    typedef U type;
};

template <class U>
U&& forward(typename identity<U>::type&& u)
{
    return u;
}
```

See Also

C++0x Features (C++Builder 2009) (see page 462)

A Proposal to Add an Rvalue Reference to the C++ Language

A Brief Introduction to Rvalue References

3.1.3.1.19.8 Static Assertions (C++0x)

The **static_assert** keyword is used to test assertions at compile time. This is one of the C++0x features added to C++Builder 2009.

This keyword operates differently than the macro **assert**, which raises assertions at run time. The keyword **static_assert** also differs from the preprocessor directive **#error**, which operates during preprocessing and simply emits a message.

Syntax

A static assertion's declaration is:

```
static_assert (constant-expression, error-message);
```

The `constant-expression` must be one that can be statically evaluated as a boolean. If `constant-expression` is true, the statement does nothing. If false, the compiler generates an error with the text `error-message`. Because the assertion is tested at compile time, **static_assert** can do error checking in templates. For instance:

```
template <class T>
   T Test(T x, T y) {
   static_assert(sizeof T <= sizeof long, "Type is too large");
   ...
   };
```

static_assert is useful for static type checking. A certain function might fail if the implementation of an int is too small, so **static_assert** has utility outside of templates.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Working Draft - Standard for Programming Language C++ (Sec. 7.1 - Specifiers)

3.1.3.1.19.9 Strongly Typed Enums (C++0x)

C++Builder 2009 introduces scoped enums. In addition, existing enums are extended with underlying type and explicit scoping. This feature is one of the C++0x features added to C++Builder 2009.

Scoped enums are generally characterized as follows:

- Enumerators are in the scope of their enum.
- Enumerators and enums do not implicitly convert to int (as do "plain" enumerators and enums).
- Enums and their enumerators can have a defined underlying type.

Declaration

You declare a scoped enum by specifying `enum class` or `enum struct`. For example:

```
enum class A {A1, A2, A3 = 50, A4 /* = 51 */};
```

No Implicit Type Conversion

With scoped enums, there is no longer any implicit conversion to or from an integer.

Underlying Type

For scoped enums, the underlying type is well specified (the default is `int`). You can specify the underlying type of the enumeration and all the enumerators by writing : type after the enumeration name (you can specify any integral type except `wchar_t`). The following example declares an enum of underlying type `unsigned long`:

```
enum class A : unsigned long {A1 = 1, A2 = 2, Abig = 0xFFFFFFFOU };
```

Scoping

A scoped enum introduces its own scope. The names of enumerators are in the enum's scope, and are not injected into the enclosing scope. For instance:

```
enum class A { A1, A2, A3 = 100, A4 /* = 101 */ };
A a1 = A1;  // error
A a2 = A::A2;   // OK-scope specified
```

Changes to Existing Enums

In addition, existing enums are being extended as follows:

- You can now specify the underlying type of any enum, just as with scoped enums (by adding `: type` to the declaration).
- Existing enums now introduce their own scopes, just as with scoped enums. The names of enumerators are defined in the enum's scope and are also injected into the enclosing scope.

For instance:

```
enum B { B1, B2, B3 = 100, B4 /* = 101 */ };
B b1 = B1;  // ok
B b2 = B::B2;   // ok
```

Examples

The following examples demonstrate the following two things:

- The method for calling scoped enumerators
- The fact that `enum class` (and `enum struct`) cannot be specified with elaborated-type-specifiers

```
enum class E { a, b };
enum E x1 = E::a; // OK
enum class E x2 = E::a; // illegal
```

See Also

C++0x Features (C++Builder 2009) (see page 462)

Strongly Typed Enums (Revision 3)

Forward Declaration of Enumerations (see page 495)

3.1.3.1.19.10 Type Specifier decltype (C++0x)

The C++0x standard includes the **decltype** keyword and operator, which represents the type of an expression. This feature is one of the C++0x features added to C++Builder 2009.

Syntax

The format of the **decltype** operator is:

```
decltype ( expression )
```

Here are the rules for evaluating `decltype(e)`:

- If e is an identifier expresson or accessing a class member, `decltype(e)` is the type of the thing designated by e. If there is no such thing, or if e is the name of a set of overloaded functions so there is ambiguity, `decltype(e)` is invalid.
- Otherwise, if e is a function call or invokes an overloaded operator, `decltype(e)` is the type returned by the function.
- Otherwise, if e is an lvalue, `decltype(e)` is a reference to `T (T&)` where T is the type of e.
- If none of the other cases apply, `decltype(e)` is the type of e.

Examples

See the embedded contents in the following examples.

```
const char *namePtr, nameChar;
decltype(namePtr);      // type is const char*
decltype(nameChar);     // type is const char
int& F(void);
decltype(F());          // type is int&

struct; D {double value; }
const D* d = new D();

decltype(d->value);    // type is double
decltype((d->value));  // type is const double&

double GetValue(int one);
long int GetValue(double d);
decltype(GetValue);     // ill-formed -- ambiguous
```

See Also

C++0x Features (C++Builder 2009) (see page 462)

Working Draft - Standard for Programming Language C++ (Sec. 7.1.6.2 - Simple Type Specifiers)

3.1.3.1.19.11 Unicode Character Types and Literals (C++0x)

C++Builder 2009 implements new character types and character literals for Unicode. These types are among the C++0x features added to C++Builder 2009.

Character Types char16_t and char32_t

Two new types represent Unicode characters:

- char16_t is a 16 bit character type. char16_t is a C++ keyword. This type could be used for UTF-16 characters.
- char32_t is a 32 bit character type. char32_t is a C++ keyword. This type can be used for UTF-32 characters.

The existing wchar_t (see page 573) type is a type for a wide character in the execution wide-character set. A wchar_t wide-character literal begins with an uppercase L (such as `L'c'`).

Character Literals u'character' and U'character'

There are two new forms to create literals of the new types:

- u'character' is a literal for a single char16_t character, such as `u'g'`. A multicharacter literal such as u'kh' is badly formed. The value of a char16_t literal is equal to its ISO 10646 code point value, provided that the code point is representable as a 16-bit cvalue. Only characters in the basic multi-lingual plane (BMP) can be represented.
- U'character' is a literal for a single char32_t character, such as `U't'`. A multicharacter literal such as U'de' is ill-formed. The value of a char32_t literal is equal to its ISO 10646 code point value.

Multibyte character literals were previously only of the form L'characters', representing one or more characters of the type wchar_t. The value of a single character wide-character literal is that character's encoding in the execution wide-character set.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Working Draft - Standard for Programming Language C++ (Sec. 2.13.2 - Character Literals)

3.1.3.1.20 Unicode for C++

This section contains Unicode related feature topics for C++.

Topics

Name	Description
TCHAR Mapping (see page 501)	**TCHAR**, which is declared in the header file *tchar.h*, is a **typedef** (alias) that maps either to **char** or to **wchar_t**. When you want to write portable code, you should use **TCHAR** instead of hard coding **char** or **wchar_t**. The **TCHAR maps to** option controls the floating definition of _TCHAR. Your choices are: ***_TCHAR Mapping Options***
Floating Functions (see page 502)	This topic describes how to write portable code that correctly uses the floating definitions of RTL functions. The list of floating functions is useful for choosing the correct variant of a function.
Enabling C++ Applications for Unicode (see page 507)	C++ has a unique set of Unicode-related issues that Delphi users do not have. Some of these issues are: • String constants, such as `"string constant"`, are still narrow (`char*`), so you cannot pass them to VCL functions that take **PChar** as you did before. If you prefix the constant with **L**, as in `L"string constant"`, you can pass the constant to VCL functions with a **PChar** parameter. • The C++ RTL is as wide and narrow as it always was. Unless you planned ahead and used <tchar.h> and the _txxxxxx() macros, you need to do some... more (see page 507)

3.1.3.1.20.1 TCHAR Mapping

TCHAR, which is declared in the header file *tchar.h*, is a **typedef** (alias) that maps either to **char** or to **wchar_t**. When you want to write portable code, you should use **TCHAR** instead of hard coding **char** or **wchar_t**.

The **TCHAR maps to** option controls the floating definition of _TCHAR. Your choices are:

_TCHAR Mapping Options

char	_TCHAR does not float to a wide definition.
wchar_t	Sets the UNICODE and _UNICODE defines. _TCHAR floats to the wide definitions of standard library and API functions.

To set the **TCHAR maps to** option, go to the **Project ▸ Options ▸ Directories and Conditionals** dialog box.

Use _TCHAR maps to wchar_t for VCL

The VCL is implemented in Unicode and always expects Unicode. To use the VCL, you should set **TCHAR maps to** to **wchar_t**. For example, the following code does not compile unless you have set _TCHAR to **wchar_t**:

```
TResourceStream* res =  new
    TResourceStream(HInstance, ResourceId, RT_RCDATA);
```

If _TCHAR is **char**, RT_RCDATA maps to a `char*`, but VCL expects **wchar_t**.

Use _TEXT function for Constants Strings

To ensure that constants strings float properly to ANSI or Unicode, use either the function _TEXT or _T. For example:

```
::MessageBox(0, _TEXT("The message"),
    _TEXT("The caption"), MB_OK);
```

Code Changes Are Required for Using _TCHAR maps to wchar_t

Before you can set the _TCHAR option to **wchar_t**, your project must have an entry point called either _tmain or _tWinMain. New projects created with C++Builder 2009 have these entry points by default, but for imported projects you might need to add these entry points by hand. You must also include the *tchar.h* header file, which contains the floating definitions and the entry

points that you need. For a list of the floating functions contained in *tchar.h*, see Floating Functions (see page 502).

See Also

Floating Functions (see page 502)

Unicode in RAD Studio

Enabling Unicode in Your Applications

Enabling C++ Applications for Unicode (see page 507)

Directories and Conditionals Dialog Box

3.1.3.1.20.2 Floating Functions

This topic describes how to write portable code that correctly uses the floating definitions of RTL functions. The list of floating functions is useful for choosing the correct variant of a function.

Example of Writing Portable Code Using _TCHAR

RTL functions are available in either ANSI/MBCS or Unicode definitions. For example, the RTL function `strlen` can be used in ANSI/MBCS mode as follows:

```
void StorePersonData(const char* name)
{
   PersistToDB(name, strlen(name));
}
```

```
A more portable way of writing this code is:
```

```
void StorePersonData(const _TCHAR* name)
{
   PersistToDB(name, _tcstrlen(name));
}
```

This example assumes that `PersistToDB` has a Unicode version. Note that for functions that are not going to change or float, you should use either **char** or **wchar_t**.

List of Floating Definitions from tchar.h

The following table is derived from the *tchar.h* header file. The **char/ANSI** column lists the floating functions that are appropriate when `_TCHAR` is mapped to `char`, the ANSI version of the function. The **wchar_t/UNICODE** column lists the functions for **wchar_t**, the **UNICODE** version of the function. The **_TCHAR/Portable** column lists the portable version of the function, which maps to the appropriate function (either the ANSI or UNICODE version) as determined by the current setting of the `_TCHAR` mapping option.

Floating Function Definitions from tchar.h

char / ANSI	wchar_t / UNICODE	_TCHAR / Portable
_access	_waccess	_taccess
_atoi64	_wtoi64	_tstoi64
_atoi64	_wtoi64	_ttoi64
_cgets	_cgetws	cgetts
_chdir	_wchdir	_tchdir
_chmod	_wchmod	_tchmod
_cprintf	_cwprintf	_tcprintf
_cputs	_cputws	_cputts

_creat	_wcreat	_tcreat
_cscanf	_cwscanf	_tcscanf
_ctime64	_wctime64	_tctime64
_execl	_wexecl	_texecl
_execle	_wexecle	_texecle
_execlp	_wexeclp	_texeclp
_execlpe	_wexeclpe	_texeclpe
_execv	_wexecv	_texecv
_execve	_wexecve	_texecve
_execvp	_wexecvp	_texecvp
_execvpe	_wexecvpe	_texecvpe
_fdopen	_wfdopen	_tfdopen
_fgetchar	_fgetwchar	_fgettchar
_findfirst	_wfindfirst	_tfindfirst
_findnext64	_wfindnext64	_tfindnext64
_findnext	_wfindnext	_tfindnext
_findnexti64	_wfindnexti64	_tfindnexti64
_fputchar	_fputwchar	_fputtchar
_fsopen	_wfsopen	_tfsopen
_fullpath	_wfullpath	_tfullpath
_getch	_getwch	_gettch
_getche	_getwche	_gettche
_getcwd	_wgetcwd	_tgetcwd
_getdcwd	_wgetdcwd	_tgetdcwd
_ltoa	_ltow	_ltot
_makepath	_wmakepath	_tmakepath
_mkdir	_wmkdir	_tmkdir
_mktemp	_wmktemp	_tmktemp
_open	_wopen	_topen
_popen	_wpopen	_tpopen
_putch	_putwch	_puttch
_putenv	_wputenv	_tputenv
_rmdir	_wrmdir	_trmdir
_scprintf	_scwprintf	_sctprintf
_searchenv	_wsearchenv	_tsearchenv
_snprintf	_snwprintf	_sntprintf
_snscanf	_snwscanf	_sntscanf
_sopen	_wsopen	_tsopen
_spawnl	_wspawnl	_tspawnl

_spawnle	_wspawnle	_tspawnle
_spawnlp	_wspawnlp	_tspawnlp
_spawnlpe	_wspawnlpe	_tspawnlpe
_spawnv	_wspawnv	_tspawnv
_spawnve	_wspawnve	_tspawnve
_spawnvp	_wspawnvp	_tspawnvp
_spawnvpe	_wspawnvpe	_tspawnvpe
_splitpath	_wsplitpath	_tsplitpath
_stat64	_wstat64	_tstat64
_stat	_wstat	_tstat
_stati64	_wstati64	_tstati64
_strdate	_wstrdate	_tstrdate
_strdec	_wcsdec	_tcsdec
_strdup	_wcsdup	_tcsdup
_stricmp	_wcsicmp	_tcsicmp
_stricoll	_wcsicoll	_tcsicoll
_strinc	_wcsinc	_tcsinc
_strlwr	_wcslwr	_tcslwr
_strncnt	_wcsncnt	_tcsnbcnt
_strncnt	_wcsncnt	_tcsnccnt
_strncnt	_wcsncnt	_tcsnccnt
_strncoll	_wcsncoll	_tcsnccoll
_strnextc	_wcsnextc	_tcsnextc
_strnicmp	_wcsnicmp	_tcsncicmp
_strnicmp	_wcsnicmp	_tcsnicmp
_strnicoll	_wcsnicoll	_tcsncicoll
_strnicoll	_wcsnicoll	_tcsnicoll
_strninc	_wcsninc	_tcsninc
_strnset	_wcsnset	_tcsncset
_strnset	_wcsnset	_tcsnset
_strrev	_wcsrev	_tcsrev
_strset	_wcsset	_tcsset
_strspnp	_wcsspnp	_tcsspnp
_strtime	_wstrtime	_tstrtime
_strtoi64	_wcstoi64	_tcstoi64
_strtoui64	_wcstoui64	_tcstoui64
_strupr	_wcsupr	_tcsupr
_tempnam	_wtempnam	_ttempnam
_ui64toa	_ui64tow	_ui64tot

_ultoa	_ultow	_ultot
_ungetch	_ungetwch	_ungettch
_unlink	_wunlink	_tunlink
_utime64	_wutime64	_tutime64
_utime	_wutime	_tutime
_vscprintf	_vscwprintf	_vsctprintf
_vsnprintf	_vsnwprintf	_vsntprintf
asctime	_wasctime	_tasctime
atof	_wtof	_tstof
atoi	_wtoi	_tstoi
atoi	_wtoi	_ttoi
atol	_wtol	_tstol
atol	_wtol	_ttol
ctime	_wctime	_tctime
fgetc	fgetwc	_fgettc
fgets	fgetws	_fgetts
fopen	_wfopen	_tfopen
fprintf	fwprintf	_ftprintf
fputc	fputwc	_fputtc
fputs	fputws	_fputts
freopen	_wfreopen	_tfreopen
fscanf	fwscanf	_ftscanf
getc	getwc	_gettc
getchar	getwchar	_gettchar
getenv	_wgetenv	_tgetenv
gets	getws	_getts
isalnum	iswalnum	_istalnum
isalpha	iswalpha	_istalpha
isascii	iswascii	_istascii
iscntrl	iswcntrl	_istcntrl
isdigit	iswdigit	_istdigit
isgraph	iswgraph	_istgraph
islower	iswlower	_istlower
isprint	iswprint	_istprint
ispunct	iswpunct	_istpunct
isspace	iswspace	_istspace
isupper	iswupper	_istupper
isxdigit	iswxdigit	_istxdigit
main	wmain	_tmain

perror	_wperror	_tperror
printf	wprintf	_tprintf
putc	putwc	_puttc
putchar	putwchar	_puttchar
puts	_putws	_putts
remove	_wremove	_tremove
rename	_wrename	_trename
scanf	wscanf	_tscanf
setlocale	_wsetlocale	_tsetlocale
sprintf	swprintf	_stprintf
sscanf	swscanf	_stscanf
strcat	wcscat	_tcscat
strchr	wcschr	_tcschr
strcmp	wcscmp	_tcscmp
strcoll	wcscoll	_tcscoll
strcpy	wcscpy	_tcscpy
strcspn	wcscspn	_tcscspn
strerror	_wcserror	_tcserror
strftime	wcsftime	_tcsftime
strlen	wcslen	_tcsclen
strlen	wcslen	_tcslen
strncat	wcsncat	_tcsncat
strncat	wcsncat	_tcsnccat
strncmp	wcsncmp	_tcsnccmp
strncmp	wcsncmp	_tcsncmp
strncpy	wcsncpy	_tcsnccpy
strncpy	wcsncpy	_tcsncpy
strpbrk	wcspbrk	_tcspbrk
strrchr	wcsrchr	_tcsrchr
strspn	wcsspn	_tcsspn
strstr	wcsstr	_tcsstr
strtod	wcstod	_tcstod
strtok	wcstok	_tcstok
strtol	wcstol	_tcstol
strtoul	wcstoul	_tcstoul
strxfrm	wcsxfrm	_tcsxfrm
system	_wsystem	_tsystem
tmpnam	_wtmpnam	_ttmpnam
tolower	towlower	_totlower

toupper	towupper	_totupper
ungetc	ungetwc	_ungettc
vfprintf	vfwprintf	_vftprintf
vprintf	vwprintf	_vtprintf
vsprintf	vswprintf	_vstprintf
WinMain	wWinMain	_tWinMain

See Also

TCHAR Mapping (see page 501)

Unicode in RAD Studio

Enabling Unicode in Your Applications

Enabling C++ Applications for Unicode (see page 507)

3.1.3.1.20.3 Enabling C++ Applications for Unicode

C++ has a unique set of Unicode-related issues that Delphi users do not have. Some of these issues are:

- String constants, such as `"string constant"`, are still narrow (`char*`), so you cannot pass them to VCL functions that take **PChar** as you did before. If you prefix the constant with **L**, as in `L"string constant"`, you can pass the constant to VCL functions with a **PChar** parameter.
- The C++ RTL is as wide and narrow as it always was. Unless you planned ahead and used <tchar.h> and the _txxxxxx() macros, you need to do some work to use the wide flavor of the RTL. VCL generally uses wide string data, so you probably need to use wide RTL functions.
- The Windows API is still narrow by default. The new _TCHAR mapping option helps trememdously here, but the option is OFF by default. You need to explicitly set the _**TCHAR maps to** option on the **Project**▶**Options**▶**Directories and Conditionals** dialog box.
- You can set the codepage for **AnsiString** types with `AnsiStringT<codepage>`. The same predefined types are available that Delphi provides:
- **AnsiString** is `AnsiStringT<0>`
- **UTF8String** is `AnsiStringT<65005>`
- **RawByteString** is `AnsiStringT<0xFFFF>`

See Also

TCHAR Mapping (see page 501)

Floating Functions (see page 502)

Unicode in RAD Studio

Enabling Unicode in Your Applications

Directories and Conditionals Dialog Box

3.1.3.1.21 Handling Delphi Features in C++Builder 2009

This section contains topics describing how you can handle issues that might arise when C++Builder 2009 interacts with certain new Delphi features.

Topics

Name	Description
How to Handle Delphi Anonymous Methods in C++ (see page 508)	This topic describes some programming issues that you might encounter when dealing with anonymous methods, one of Delphi's newest features. Under the cover, Delphi implements anonymous methods types (also known as method references) via an interface that implements an `Invoke(...)` method. So a method that takes a method reference parameter in Delphi is exposed to C++ as a method that takes an interface. Here's an example:
How to Handle Delphi AnsiString Code Page Specification in C++ (see page 510)	The C++ implementation of the AnsiString type provides CodePage support similar to that available in Delphi 2009. However, while the Delphi version is implemented via extension to the language, the C++ version is implemented via a template with a non-type parameter as codepage, as in:
How to Handle Delphi Generics in C++ (see page 511)	This topic describes some programming issues that you might encounter when dealing with generics, one of Delphi's newest features. Delphi generics are exposed to C++ as templates. However, it's important to realize that the instantiations occur on the Delphi side, not in C++. Therefore, you can only use these template for types that were explicitly instantiated in Delphi code. For example, let's declare a simply generic, `TList<T>`, in Delphi:

3.1.3.1.21.1 How to Handle Delphi Anonymous Methods in C++

This topic describes some programming issues that you might encounter when dealing with anonymous methods, one of Delphi's newest features.

Under the cover, Delphi implements anonymous methods types (also known as method references) via an interface that implements an `Invoke(...)` method. So a method that takes a method reference parameter in Delphi is exposed to C++ as a method that takes an interface. Here's an example:

```
interface

type

  TRefProc = reference to function (I, J: Integer): Integer;

  TTestClass = class
    public
      function TakeRefProc(RefProc: TRefProc; I, J: Integer): Integer;
  end;
```

The following is the .hpp file generated for the above:

```
__interface TRefProc;
typedef System::DelphiInterface <TRefProc> _di_TrefProc;

__interface TRefProc  : public System::IInterface
{
public:
 virtual int __fastcall Invoke(int I, int J) = 0 ;
};

class PASCALIMPLEMENTATION TTestClass : public System::TObject
{
public:
 int __fastcall TakeRefProc(_di_TRefProc RefProc, int I, int J);
};
```

C++ code that seeks to specify a function or member function as a method reference parameter has to wrap the latter behind an interface that exposes an `Invoke()` method. A C++ template can be used to encapsulated such an interface. The following C++ code shows an example of a template that can be used to pass C++ methods or member functions as method references to Delphi.

```
enum _DummyType{};      // Parameter used as default

template <typename INTF,                        // Interface with Invoke
```

```
                typename F,                            // Function type
                typename R,                           // Return type
                typename P1 = _DummyType,  // Param #1
                typename P2 = _DummyType,  // Param #2
                typename P3 = _DummyType,  // Param #3
                typename P4 = _DummyType,  // Param #4
                typename P5 = _DummyType> // Param #5
class TMethodRef : public TInterfacedObject, public INTF

{
private:
  F callback;
public:
  TMethodRef(F _callback) : callback(_callback) {}

  HRESULT STDMETHODCALLTYPE QueryInterface (const GUID& riid, void** ppvObject)
  { return TInterfacedObject::QueryInterface (riid, ppvObject); }
  ULONG STDMETHODCALLTYPE AddRef()
  { return TInterfacedObject::_AddRef(); }
  ULONG STDMETHODCALLTYPE Release()
  { return TInterfacedObject::_Release(); }

R __fastcall Invoke(P1 p1)
  {
    return callback(p1);
  }
  R __fastcall Invoke(P1 p1, P2 p2)
  {
    return callback(p1, p2);
  }
  R __fastcall Invoke(P1 p1, P2 p2, P3 p3)
  {
    return callback(p1, p2, p3);
  }
  R __fastcall Invoke(P1 p1, P2 p2, P3 p3, P4 p4)
  {
    return callback(p1, p2, p3, p4);
  }
  R __fastcall Invoke(P1 p1, P2 p2, P3 p3, P4 p4, P5 p5)
  {
    return callback(p1, p2, p3, p4, p5);
  }
};
```

The following code shows how to use the template shown above to pass a C++ routine as a method reference.

```
// C++ function we want to pass as a method reference.
int multiplyCallback(int i, int j)
{
  return i*j;
}


void UseRefProcFlat()
{
  std::auto_ptr<TTestClass> cls(new TTestClass());
  _di_TRefProc proc = new TMethodRef<TRefProc, int (*)(int, int), int, int,
int>(multiplyCallback);
  int i = cls->TakeRefProc(proc, 10, 20);
  assert(i == 200);
}
```

You can also use the template to pass a member function as a method reference. The following code illustrates this:

```
class TMyClass {
public:
  int add(int i, int j) {
```

```
    return i+j;
  }
};
typedef int (__closure *TClosure)(int, int);

void UseRefProcMember()
{
  TMyClass myClass;

  std::auto_ptr<TTestClass> cls(new TTestClass());
  _di_TRefProc proc = new TMethodRef<TRefProc, TClosure, int, int, int>(&myClass.add);

  int i = cls->TakeRefProc(proc, 10, 20);
  assert(i == 30);
}
```

See Also

Handling Generics (see page 511)

Handling AnsiString Code Page Specification (see page 510)

3.1.3.1.21.2 How to Handle Delphi AnsiString Code Page Specification in C++

The C++ implementation of the AnsiString type provides CodePage support similar to that available in Delphi 2009. However, while the Delphi version is implemented via extension to the language, the C++ version is implemented via a template with a non-type parameter as codepage, as in:

```
template AnsiStringT<unsigned short codePage>
```

Therefore, the type previously known as AnsiString type is now simply a typedef of the new AnsiStringT template, as in:

```
 typedef AnsiStringT<65001> UTF8String;

 typedef AnsiStringT<65535> RawByteString;
```

You can declare other specializations of AnsiStringT with the codepage you need.

All assignments to an AnsiStringT instance will encode the data in the type's codepage. For example, in the following code, the Unicode data assigned to the UTF8String variable is automatically UTF8-encoded:

```
const wchar_t* data = L"???????";
UTF8String utfs(data);
```

The 'utfs' variable can be passed to a function expecting a UnicodeString and the original data will be restored without any loss incurred:

```
Button1->Caption = utfs; // Set button caption to "???????";
```

Note that while the AnsiStringT handles the codepage support behind the scene, you can still explicitly set the codepage of an instance by calling the following method:

```
AnsiStringT<CP>::SetCodePage(unsigned short codePage, bool
    convert=true)
```

This method should be used carefully because it might mislead users of the instance who expect the type to contain data encoded in its default codepage.

See Also

Handling Anonymous Methods (see page 508)

Handling Generics (see page 511)

3.1.3.1.21.3 How to Handle Delphi Generics in C++

This topic describes some programming issues that you might encounter when dealing with generics, one of Delphi's newest features.

Delphi generics are exposed to C++ as templates. However, it's important to realize that the instantiations occur on the Delphi side, not in C++. Therefore, you can only use these template for types that were explicitly instantiated in Delphi code. For example, let's declare a simply generic, `TList<T>`, in Delphi:

```
 interface

type

    MyTList<T> = class(TList<T>) // TList is a class in the Generics.Collections namespace
        FItems: array of T;
    protected
        function GetLength: Integer;
    public
        function Get(Index: Integer): T;
    published
        property Len: Integer read GetLength;
    end;

    { ScoreList derived from a TList<double>}
    ScoreList = class(MyTList<double>)
    end;
    { StringList derived from a TList<string>}
    StringList = class(MyTList<string>)
    end;

implementation

{$R *.dfm}

function MyTList<T>.GetLength: Integer;
begin
    Result := Count;
end;

  function MyTList<T>.Get(Index: Integer): T;
begin
    Result := Items[Index];
end;
```

The interface above is exposed to C++ as the following:

```
// Template declaration generated by Delphi parameterized types is
// used only for accessing Delphi variables and fields.
// Don't instantiate with new type parameters in user code.
template<typename T> class PASCALIMPLEMENTATION MyTList__1 : public
Generics_collections::TList__1<T>
{
typedef Generics_collections::TList__1<T> inherited;

private:
typedef DynamicArray<T> _MyTList__1__1;

public:
    _MyTList__1__1 FItems;

 protected:
    int __fastcall GetLength(void);

public:
    T __fastcall Get(int Index);
```

```
__published:
     __property int Len = {read=GetLength, nodefault};
public:
/* TList<T>.Create */ inline __fastcall MyTList__1(void)/* overload */ :
Generics_collections::TList__1<T>() { }
      /* TList<T>.Destroy */ inline __fastcall virtual ~MyTList__1(void) { }

};

class DELPHICLASS ScoreList;
class PASCALIMPLEMENTATION ScoreList : public MyTList__1<double>
{
typedef MyTList__1<double> inherited;

public:
/* TList<Double>.Create */ inline __fastcall ScoreList(void)/* overload */ :
MyTList__1<double>() { }
      /* TList<Double>.Destroy */ inline __fastcall virtual ~ScoreList(void) { }

};

class DELPHICLASS StringList;
class PASCALIMPLEMENTATION StringList : public MyTList__1<System::UnicodeString>
{
typedef MyTList__1<System::UnicodeString> inherited;

public:
/* TList<string>.Create */ inline __fastcall StringList(void)/* overload */ :
MyTList__1<System::UnicodeString>() { }
/* TList<string>.Destroy */ inline __fastcall virtual ~StringList(void) { }

};
```

C++ code linking with the .obj created from the above Delphi unit can use instances of `TList__1<double>` or `ScoreList`.

```
void UseScoreList()
{
  ScoreList* list = new ScoreList();

  list->Add(1.0);
  list->Add(2.0);

  int len = list->Len;
  assert(len == 2);

  delete list;
}

void UseTList__1()


{
    // C++ code can use the Generics defined in Delphi directly
    // as long as the C++ code limits itself to types for which
    // the generic was instantiated on the Delphi size. For example,
    // since test.pas uses TList<String> and TList<double> we can use
    // these here. However, if we try to use TList__1>char> we'll get
    // an error since the Delphi side did not instantiate
    // TList<AnsiChar>.
    TList__1<double>* dblList = new MyTList__1<double>();
    dblList—>Add(1.0);
    dblList—>Add(1.5);
    double d = dblList—>Get(1);
    delete dblList;
```

```
    MyTList__1<UnicodeString> *stringList = new MyTList__1<UnicodeString>();
    stringList->Add("hiya");
    stringList->Add("there");
    stringList->Add("buckeroo");
    UnicodeString dstring = stringList->Get(0);
    delete stringList;
 }
```

If C++ code attempts to use a Delphi generic for types that were not instantiated in Delphi, you'll get errors at link time. For example, the following code attempts to use `TList__1<char>` when the Delphi code did not explicitly instantiate `TList<AnsiChar>`:

```
void UseListOfChar()
{
  TList__1<char>* charList = new TList__1<char>();
charList->Add('a');
  char ch = charList->Get(1);
  delete charList;
}
```

While the code above compiles, the following errors are generated at link time:

```
[ILINK32 Error] Error: Unresolved external 'Test::MyTList__1<char>::>::' referenced from
USETEST.OBJ
[ILINK32 Error] Error: Unresolved external '__fastcall Test::MyTList__1<char>>::Add(char)'
referenced from USETEST.OBJ
[ILINK32 Error] Error: Unresolved external '__fastcall Test::MyTList__1<char>::Get(int)'
referenced from USETEST.OBJ
```

To eliminate the error, you have to make sure that the Delphi code uses the type `MyTList<AnsiChar>`.

See Also

Handling Anonymous Methods (see page 508)

Handling AnsiString Code Page Specification (see page 510)

3.1.3.2 Keywords, Alphabetical Listing

This section contains Keywords, Alphabetical Listing topics.

Topics

Name	Description
__automated (see page 522)	**Category** Keyword extensions **Syntax**
__classid (see page 523)	**Category** Operators, Keyword extensions **Syntax**
__closure (see page 523)	**Category** Keyword extensions **Syntax**
__declspec (see page 523)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
__dispid (see page 525)	**Category** Modifiers **Syntax**
__except (see page 525)	**Category** Statements, Keyword extensions **Syntax**

__inline (see page 525)	**Category** Keyword extensions **Syntax**
__int8, __int16, __int32, __int64, Unsigned __int64, Extended Integer Types (see page 526)	**Category** Keyword extensions **Description** You can specify the size for integer types. You must use the appropriate suffix when using extended integers.
__msfastcall (see page 526)	**Category** Modifiers, Keyword extensions **Syntax**
__msreturn (see page 527)	**Category** Modifiers, Keyword extensions **Syntax**
__property (see page 527)	**Category** Keyword extensions **Syntax**
__published (see page 528)	**Category** Keyword extensions **Syntax**
__rtti, -RT Option (see page 528)	**Category (__rtti keyword)** Modifiers, C++ Keyword Extensions, C++-Specific Keywords **Description** Runtime type identification is enabled by default. You can disable RTTI on the C++ page of the Project Options dialog box. From the command-line, you can use the **-RT-** option to disable it or **-RT** to enable it. If RTTI is disabled, or if the argument to **typeid** is a pointer or a reference to a non-polymorphic class, **typeid** returns a reference to a **const** type_info object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.... more (see page 528)
__thread, Multithread Variables (see page 529)	**Category** Keyword extensions **Description** The keyword __**thread** is used in multithread programs to preserve a unique copy of global and static class variables. Each program thread maintains a private copy of a __**thread** variable for each thread. The syntax is Type __**thread** variable__name. For example
__try (see page 529)	**Category** Statements, Keyword extensions **Syntax**
_export, __export (see page 530)	**Category** Modifiers, Keyword extensions **Form 1**
_fastcall, __fastcall (see page 530)	**Category** Modifiers, Keyword extensions **Syntax**
_stdcall, __stdcall (see page 531)	**Category** Modifiers, Keyword extensions **Syntax**
alignof (see page 531)	**Category** Operators, Keyword extensions **Syntax**

and, && (see page 531)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **and** operator is an alternative representation of the **&&** operator (binary or logical AND). If both operands have a value of true, the result has the value true. Otherwise, the result has the value false. Both operands are implicitly converted to bool and the result type is bool. Unlike the **&** (bitwise AND) operator, the **&&** operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to 0 (or false), the right operand is not evaluated. Both the **and** and **&&** operators short-circuit (that is, do not... more (see page 531)
asm, _asm, __asm (see page 532)	**Category** Keyword extensions, C++-Specific Keywords **Syntax**
auto (see page 533)	**Category** Storage class specifiers **Syntax**
bitand, & (see page 533)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **bitand** operator is an alternative representation of the **&**operator (bitwise AND). The bitwise AND operator compares each bit of the first operand to the corresponding bit of the second operand When both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0. In order to use the **bitand** operator, you need to check the **Enable new operator names** option (the **-Vn** compiler switch, available on the **Compatibility** page of the **Project** ▶ **Options** dialog box).
bitor, | (see page 533)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **bitor** operator is an alternative representation of the **|~**operator (bitwise OR). **bitor** takes integer or enumeration operands. **bitor** may pass two or more arguments. These arguments must be character or numeric. When coded as a keyword, **bitor** can only pass two arguments. In either case, the result type is the same as the arguments: (1) unsigned if all arguments are unsigned, or (2) an integer. In order to use the **bitor** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available... more (see page 533)
bool, false, true (see page 533)	**Category** C++-Specific Keywords **Syntax**
break (see page 534)	**Category** Statements **Syntax** **Description** Use the **break** statement within loops to pass control to the first statement following the innermost **switch**, **for**, **while**, or **do** block. **Example** This example illustrates the use of keywords **break**, **case**, **default**, **return**, and **switch**.
case (see page 535)	**Category** Statements **Syntax**
catch (see page 536)	**Category** Statements, C++-Specific Keywords **Syntax**

cdecl, _cdecl, __cdecl (see page 536)	**Category** Modifiers, Keyword extensions **Syntax**
char (see page 536)	**Category** Type specifiers **Syntax**
char16_t (see page 537)	**Category** C++-Specific Keywords, Type specifiers **Syntax**
char32_t (see page 537)	**Category** C++-Specific Keywords, Type specifiers **Syntax**
class (see page 537)	**Category** C++-Specific Keywords, Type specifiers **Syntax**
compl, ~ (see page 538)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **compl** operator is an alternative representation of the **~**operator (bitwise NOT). The **compl** operator is the abbreviation for complement. **compl** Inverts true to false and false to true, The tilde (~) is placed in front of the integer used for the complement. The complement of 1 would be 0, and vice versa. Besides an integer, ~ also outputs an enumeration type; this can be the result of the one's complement of the operand. Integral calculations are performed. The type of the result is the type of the... more (see page 538)
const (see page 538)	**Category** Modifiers **Syntax**
const_cast (typecast Operator) (see page 539)	**Category** C++-Specific Keywords **Syntax**
continue (see page 540)	**Category** Statements **Syntax**
__declspec(dllexport) (see page 540)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
__declspec(dllimport) (see page 541)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
__declspec(naked) (see page 541)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
__declspec(noreturn) (see page 541)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
__declspec(nothrow) (see page 542)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
__declspec(novtable) (see page 542)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
__declspec(property) (see page 543)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
__declspec(selectany) (see page 543)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**

__declspec(thread) (see page 544)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
__declspec(uuid("ComObjectGUID")) (see page 544)	**Category** Modifiers, Keyword extensions, Storage class specifiers **Syntax**
decltype (see page 545)	**Category** Type specifiers **Syntax**
default (see page 545)	**Category** Statements **Syntax**
delete (see page 546)	**Category** Operators, C++-Specific Keywords **Syntax**
do (see page 546)	**Category** Statements **Syntax**
double (see page 547)	**Category** Type specifiers **Syntax**
dynamic_cast (typecast Operator) (see page 547)	**Category** C++-Specific Keywords **Description** In the expression, dynamic_cast< T > (ptr), T must be a pointer or a reference to a defined class type or **void***. The argument ptr must be an expression that resolves to a pointer or reference. If T is void* then ptr must also be a pointer. In this case, the resulting pointer can access any element of the class that is the most derived element in the hierarchy. Such a class cannot be a base for any other class. Conversions from a derived class to a base class, or from one derived class to... more (see page 547)
enum (see page 548)	**Category** Type specifiers **Syntax**
explicit (see page 548)	**Category** C++-Specific Keywords **Syntax**
export (see page 549)	**Category** Unimplemented **Syntax**
extern (see page 549)	**Category** Storage class specifiers **Syntax**
final (see page 550)	**Category** Attributes, Keyword extensions **Syntax**
__finally (see page 550)	**Category** Statements, Keyword extensions **Syntax**
float (see page 552)	**Category** Type specifiers **Syntax**
for (see page 552)	**Category** Statements **Syntax**
friend (see page 553)	**Category** C++-Specific Keywords **Syntax**

goto (see page 553)	**Category** Statements **Syntax**
if, else (see page 553)	**Category** Operators **C++ Syntax**
import, _import, __import (see page 554)	**Category** Modifiers, Keyword extensions **Form 1**
inline (see page 554)	**Category** C++-Specific Keywords **Syntax**
int (see page 555)	**Category** Type specifiers **Syntax**
long (see page 555)	**Category** Type specifiers **Syntax**
mutable (see page 555)	**Category** C++-Specific Keywords, Storage class specifiers **Syntax**
namespace (see page 556)	**Category** C++-Specific Keywords **Description** Most real-world applications consist of more than one source file. The files can be authored and maintained by more than one developer. Eventually, the separate files are organized and linked to produce the final application. Traditionally, the file organization requires that all names that aren't encapsulated within a defined namespace (such as function or class body, or translation unit) must share the same global namespace. Therefore, multiple definitions of names discovered while linking separate modules require some way to distinguish each name. The solution to the problem of name clashes in the global scope is provided... more (see page 556)
new (see page 556)	**Category** Operators, C++-Specific Keywords **Syntax**
noreturn (see page 557)	**Category** Attributes, Keyword extensions **Syntax**
not, ! (see page 557)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **not** operator is an alternative representation of the **!**operator (logical negation). **not** returns *true* if its operand is *false*, and *false* if its operand is *true*. If an integer is 1, the expression of !1 indicates that the number is other than 1 So that is a true statement. **not** can also be used for strings as well. The expression !Smith indicates that the person's name is other than Smith.. **not** inverts the bits of the expression. In order to use the **not** operator, you need... more (see page 557)
not_eq, != (see page 558)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **not_eq** operator is an alternative representation of **!=** (bitwise inequality). It tests for logical inequivalence. **not_eq** compares two expressions to determine whether or not they are the same. Therefore, 7 != 8 returns true, while 7 != 7 returns false. The same is true for any expression such as Smith != Smith. which returns false. In order to use the **not_eq** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project** ▶ **Options** dialog box).

nullptr (see page 558)	**Category** Reserved Words **Syntax**
operator (see page 558)	**Category** Operators, C++-Specific Keywords **Syntax**
or, || (see page 559)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **or** operator is an alternative representation of the **||** operator (logical OR). Only two statements can be evaluated at a time. **or** returns true if two values are different, such as 2 or 6. **or** returns false if both values are the same, such as 10 and 10. In order to use the **or** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project** ▶ **Options** dialog box).
or_eq, |= (see page 559)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **or_eq** operator is an alternative representation of **|=**(bitwise inclusive OR). **or_eq** tests for logical equivalence or bitwise equality. **or_eq** operates on two values. When the first value or the second value is 1, true is returned. When the values are 0, false is returned. In order to use the **not_eq** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project** ▶ **Options** dialog box).
pascal, _pascal, __pascal (see page 559)	**Category** Modifiers, Keyword extensions **Syntax**
private (see page 560)	**Category** C++-Specific Keywords **Syntax**
protected (see page 560)	**Category** C++-Specific Keywords **Syntax**
public (see page 561)	**Category** C++-Specific Keywords **Syntax**
register (see page 561)	**Category** Storage class specifiers **Syntax**
reinterpret_cast (typecast Operator) (see page 561)	**Category** C++-Specific Keywords **Syntax**
return (see page 562)	**Category** Statements **Syntax**
short (see page 563)	**Category** Type specifiers **Syntax**
signed (see page 563)	**Category** Type specifiers **Syntax**

sizeof (see page 563)	**Category** Operators **Description** The **sizeof** operator has two distinct uses: **sizeof** unary-expression **sizeof** (type-name) The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). The amount of space that is reserved for each type depends on the machine. In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type **char** (**signed** or **unsigned**), **sizeof** gives the result 1. When the... more (see page 563)
static (see page 564)	**Category** Storage class specifiers **Syntax**
static_assert (see page 565)	**Category** Statements, C++-Specific Keywords **Syntax**
static_cast (typecast Operator) (see page 565)	**Category** C++-Specific Keywords **Syntax**
struct (see page 566)	**Category** Type specifiers **Syntax**
switch (see page 567)	**Category** Statements **Syntax**
template (see page 568)	**Category** C++-Specific Keywords **Syntax**
this (see page 568)	**Category** C++-Specific Keywords **Example**
throw (see page 569)	**Category** Statements, C++-Specific Keywords **Syntax**
try (see page 569)	**Category** Statements, C++-Specific Keywords **Syntax**
typedef (see page 569)	**Category** Storage class specifiers **Syntax**
typeid (see page 570)	**Category** Operators, C++-Specific Keywords **Syntax**
typename (see page 570)	**Category** C++-Specific Keywords **Syntax 1**
union (see page 571)	**Category** Type specifiers **Syntax**
unsigned (see page 571)	**Category** Type specifiers **Syntax**

using (declaration) (see page 572)	**Category** C++-Specific Keywords **Description** You can access namespace members individually with the using-declaration syntax. When you make a using declaration, you add the declared identifier to the local namespace. The grammar is using-declaration: **using ::** unqualified-identifier;
virtual (see page 572)	**Category** C++-Specific Keywords **Syntax**
void (see page 572)	**Category** Special types **Syntax**
volatile (see page 573)	**Category** Modifiers **Syntax**
wchar_t (see page 573)	**Category** C++-Specific Keywords, Type specifiers **Syntax**
while (see page 574)	**Category** Statements **Syntax**
xor, ^ (see page 574)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **xor** operator is an alternative representation of the **^** operator (bitwise xor). It returns a Boolean true result if just one of its operands is true. This is in opposition to an inclusive **or** which denotes that both statements must be integers for a true statement to be returned. If 2 or 8.25 are stated to be integers, a true statement will be returned even though 8.25 is a decimal. If Jack and Jill are both stated to be male, a true statement would be returned even though Jill... more (see page 574)
__classmethod (see page 574)	**Category** Modifiers, Keyword extensions **Syntax**
alignas (see page 575)	**Category** C++-Specific Keywords **Syntax**
and_eq, &= (see page 575)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **and_eq** operator is an alternative representation of the **&=** assignment operator (bitwise AND). The value of the first operand is added to the value of the second operand, and the result is stored in the first operand. In order to use the **and_eq** operator, you need to check the **Enable new operator names** option (the **-Vn** compiler switch, available on the **Compatibility** page of the **Project** ▶ **Options** dialog box).
axiom (see page 575)	**Category** Reserved Words **Syntax**
concept (see page 576)	**Category** Reserved Words **Syntax**
concept_map (see page 576)	**Category** Reserved Words **Syntax**

constexpr (see page 576)	**Category** Reserved Words **Syntax**
late_check (see page 576)	**Category** Reserved Words **Syntax**
requires (see page 577)	**Category** Reserved Words **Syntax**
thread_local (see page 577)	**Category** Reserved Words **Syntax**
xor_eq, ^= (see page 577)	**Category** Alternative Representations of Operators and Tokens, Operators **Syntax** **Description** The **xor_eq** operator is an alternative representation of the **^=;** operator (bitwise XOR assignment).. True is returned if one number on the left side of an equation is the same as another number on the right side, such as 10*5!=10*2. False is returned even if 6*3=9*2. In order to use the **xor** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project** ▶ **Options** dialog box).
_Bool (see page 578)	**Category** Reserved Words **Syntax**
_Complex (see page 578)	**Category** Reserved Words **Syntax**
_Imaginary (see page 578)	**Category** Reserved Words **Syntax**
restrict (see page 579)	**Category** Reserved Words **Syntax**

3.1.3.2.1 __automated

Category

Keyword extensions

Syntax

```
_automated: <declarations>
```

Description

The visibility rules for automated members are identical to those of public members. The only difference between automated and public members is that OLE automation information is generated for member functions and properties that are declared in an automated section. This OLE automation type information makes it possible to create OLE Automation servers.

- For a member function, the types of all member function parameters and the function result (if any) must be automatable. Likewise, for a property, the property type and the types of any array property parameters must be automatable. The automatable types are: Currency, OleVariant, DelphiInterface, double, int, float, short, String, TDateTime, Variant, and unsigned short. Declaring member functions or properties that use non-automatable types in an __automated section results in a compile-time error.
- Member function declarations must use the __fastcall calling convention.
- Member functions can be virtual.

- Member functions may add __dispid(constant int expression) after the closing parenthesis of the parameter list.
- Property declarations can only include access specifiers (__dispid, read, and write). No other specifiers (index, stored, default, nodefault) are allowed.
- Property access specifiers must list a member function identifier. Data member identifiers are not allowed.
- Property access member functions must use the __fastcall calling convention.
- Property overrides (property declarations that don't include the property type) are not allowed.

See Also

__dispid (see page 525)

__closure (see page 523)

__property (see page 527)

__published (see page 528)

3.1.3.2.2 __classid

Category

Operators, Keyword extensions

Syntax

```
__classid(classType)
```

Description

The __**classid** operator was added to support the VCL framework. Normally, programmers should not directly use this operator. For more information, see the keyword extensions.

See Also

__classmethod (see page 574)

__property (see page 527)

3.1.3.2.3 __closure

Category

Keyword extensions

Syntax

```
<type> ( __closure * <id> ) (<param list>);
```

Description

The keyword __**closure** was added to support the VCL framework and is used when declaring event handler functions. For more information, see the keyword extensions.

See Also

__property (see page 527)

3.1.3.2.4 __declspec

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec(<decl-modifier>)
```

Description

For a list of **__declspec** keyword arguments used for the VCL framework, see VCL class declarations.

Use the **__declspec** keyword to indicate the storage class attributes for a variable or function.

The **__declspec** keyword extends the attribute syntax for storage class modifiers so that their placement in a declarative statement is more flexible. The **__declspec** keyword and its argument can appear anywhere in the declarator list, as opposed to the old-style modifiers which could only appear immediately preceding the identifier to be modified.

```
__export void f(void);                      // illegal
void __export f(void)                       // correct
void __declspec(dllexport) f(void);         // correct
__declspec(dllexport)void f(void);          // correct
class __declspec(dllexport) ClassName { }   // correct
```

In addition to the arguments listed above, the supported decl-modifier arguments are:

- dllexport
- dllimport
- naked
- noreturn
- nothrow
- novtable
- property
- selectany
- thread
- uuid

These arguments are equivalent to the following storage class attribute keywords.

Argument	Comparable keyword
dllexport	__export
dllimport	__import
thread	__thread

See Also

_export (see page 530)

import (see page 554)

__declspec(dllexport) (see page 540)

__declspec(dllimport) (see page 541)

__declspec(naked) (see page 541)

__declspec(noreturn) (see page 541)

__declspec(nothrow) (see page 542)

__declspec(novtablet) (see page 542)

__declspec(property) (see page 543)

__declspec(selectany) (see page 543)

__declspec(thread) (see page 544)

__declspec(uuid) (see page 544)

3.1.3.2.5 __dispid

Category

Modifiers

Syntax

```
__dispid(constant int expression)
```

Description

A member function that has been declared in the __automated section of a class can include an optional __dispid(constant int expression) directive. The directive must be declared after the closing parenthesis of the parameter list.

The constant int expression gives the Automation dispatch ID of the member function or property. If a dispid directive is not used, the compiler automatically picks a number one larger than the largest dispatch ID used by any member function or property in the class and its base classes.

Specifying an already-used dispatch ID in a dispid directive causes a compile-time error.

See Also

__closure (see page 523)

__property (see page 527)

3.1.3.2.6 __except

Category

Statements, Keyword extensions

Syntax

```
__except (expression) compound-statement
```

Description

The __except keyword specifies the action that should be taken when the exception specified by expression has been raised.

See Also

__finally (see page 550)

__try (see page 529)

3.1.3.2.7 __inline

Category

Keyword extensions

Syntax

```
__inline <datatype> <class>_<function> (<parameters>) { <statements>; }
```

Description

Use the **__inline** keyword to declare or define C or C++ inline functions. The behavior of the **__inline** keyword is identical to that of the **inline** keyword, which is only supported in C++.

Inline functions are best reserved for small, frequently used functions.

3.1.3.2.8 __int8, __int16, __int32, __int64, Unsigned __int64, Extended Integer Types

Category

Keyword extensions

Description

You can specify the size for integer types. You must use the appropriate suffix when using extended integers.

Type	Suffix	Example	Storage
__int8	i8	__int8 c = 127i8;	8 bits
__int16	i16	__int16 s = 32767i16;	16 bits
__int32	i32	__int32 i = 123456789i32;	32 bits
__int64	i64	__int64 big = 12345654321i64;	64 bits
unsigned __int64	ui64	unsigned __int64 hugeInt = 1234567887654321ui64;	64 bits

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

3.1.3.2.9 __msfastcall

Category

Modifiers, Keyword extensions

Syntax

```
__msfastcall <function-name>
```

Description

This calling convention emulates the Microsoft implementation of the fastcall calling convertion. The first two DWORD or smaller arguments are passed in ECX and EDX registers, all other arguments are passed from right to left. The called function is responsible for removing these arguments from the stack.

3.1.3.2.10 __msreturn

Category

Modifiers, Keyword extensions

Syntax

```
__msreturn <function-name>
```

Description

This calling convention is used for Microsoft compatible __**fastcall** calling convention return values. Structures with a size that is greater than 4 bytes and less than 9 bytes, and having at least one of its members sized 4 bytes or larger, are returned in EAX/EDX.

3.1.3.2.11 __property

Category

Keyword extensions

Syntax

```
<property declaration> ::=
     __property <type> <id> [ <prop dim list> ] = "{" <prop attrib list> "}"

<prop dim list> ::=
     "[" <type> [ <id> ] "]" [ <prop dim list> ]

<prop attrib list> ::=
     <prop attrib> [ , <prop attrib list> ]

<prop attrib> ::=
     read = <data/function id> |
     write = <data/function id> |
     stored = <data/function id> |
     stored = <boolean constant> |
     default = <constant> |
     nodefault |
     index = <const int expression>
```

Description

The __**property** keyword was added to support the VCL

See Also

__closure (see page 523)

3.1.3.2.12 __published

Category

Keyword extensions

Syntax

```
__published: <declarations>
```

Description

The **__published** keyword was added to support the VCL.

See Also

__closure (see page 523)

__dispid (see page 525)

3.1.3.2.13 __rtti, -RT Option

Category (__rtti keyword)

Modifiers, C++ Keyword Extensions, C++-Specific Keywords

Description

Runtime type identification is enabled by default. You can disable RTTI on the C++ page of the Project Options dialog box. From the command-line, you can use the **-RT-** option to disable it or **-RT** to enable it.

If RTTI is disabled, or if the argument to **typeid** is a pointer or a reference to a non-polymorphic class, **typeid** returns a reference to a **const** type_info object that describes the declared type of the pointer or reference, and not the actual object that the pointer or reference is bound to.

In addition, even when RTTI is disabled, you can force all instances of a particular class and all classes derived from that class to provide polymorphic runtime type identification (where appropriate) by using the **__rtti** keyword in the class definition.

When runtime type identification is disabled, if any base class is declared **__rtti**, then all polymorphic base classes must also be declared **__rtti**.

```
struct __rtti S1 { virtual s1func(); };  /* Polymorphic */
struct __rtti S2 { virtual s2func(); };  /* Polymorphic */
struct X : S1, S2 { };
```

If you turn off the RTTI mechanism, type information might not be available for derived classes. When a class is derived from multiple classes, the order and type of base classes determines whether or not the class inherits the RTTI capability.

When you have polymorphic and non-polymorphic classes, the order of inheritance is important. If you compile the following declarations without RTTI, you should declare X with the **__rtti** modifier. Otherwise, switching the order of the base classes for the class X results in the compile-time error "Can't inherit non-RTTI class from RTTI base 'S1'."

```
struct __rtti S1 { virtual func(); };   /* Polymorphic class */
struct S2 { };                           /*  Non-polymorphic class */
struct __rtti X : S1, S2 { };
```

Note: The class X is explicitly declared with __rtti

. This makes it safe to mix the order and type of classes. In the following example, class X inherits only non-polymorphic classes. Class X does not need to be declared **__rtti**.

```
struct __rtti S1 {  };   // Non-polymorphic class
```

```
struct S2 { };
struct X : S2, S1 { };   // The order is not essential
```

Neither the __**rtti** keyword, nor enabling RTTI will make a static class into a polymorphic class.

See Also

Runtime Type Identification (RTTI) Overview (see page 453)

Runtime Type Identification And Destructors (see page 454)

3.1.3.2.14 __thread, Multithread Variables

Category

Keyword extensions

Description

The keyword __**thread** is used in multithread programs to preserve a unique copy of global and static class variables. Each program thread maintains a private copy of a __**thread** variable for each thread.

The syntax is Type __**thread** variable__name. For example

```
int __thread x;
```

This statement declares an integer type variable that will be global but private to each thread in the program in which the statement occurs.

3.1.3.2.15 __try

Category

Statements, Keyword extensions

Syntax

```
__try compound-statement handler-list__try
__try compound-statement termination-statement
```

Description

The __**try** keyword is supported in both C and C++ programs. You can also use try in C++ programs.

A block of code in which an exception can occur must be prefixed by the keyword __**try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the normal program flow is interrupted. The program begins a search for a handler that matches the exception. If the exception is generated in a C module, it is possible to handle the structured exception in either a C module or a C++ module.

If a handler can be found for the generated structured exception, the following actions can be taken:

- Execute the actions specified by the handler
- Ignore the generated exception and resume program execution
- Continue the search for some other handler (regenerate the exception)

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

See Also

catch (see page 536)

__finally (see page 550)

3.1.3.2.16 _export, __export

Category

Modifiers, Keyword extensions

Form 1

```
class _export <class name>
```

Form 2

```
return_type _export <function name>
```

Form 3

```
data_type _export <data name>
```

Description

These modifiers are used to export classes, functions, and data.

The linker enters functions flagged with **_export** or **__export** into an export table for the module.

Using **_export** or **__export** eliminates the need for an EXPORTS section in your module definition file.

Functions that are not modified with **_export** or **__export** receive abbreviated prolog and epilog code, resulting in a smaller object file and slightly faster execution.

Note: If you use _export

or **__export** to export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient). If you want to change various attributes from the default, you'll need a module definition file.

3.1.3.2.17 _fastcall, __fastcall

Category

Modifiers, Keyword extensions

Syntax

```
return-value _fastcall function-name(parm-list)
return-value __fastcall function-name(parm-list)
```

Description

Use the **__fastcall** modifier to declare functions that expect parameters to be passed in registers. The first three parameters are passed (from left to right) in EAX, EDX, and ECX, if they fit in the register. The registers are not used if the parameter is a floating-point or struct type.

Note: All VCL class member functions and form class member functions must use the __fastcall

convention. The compiler treats this calling convention as a new language specifier, along the lines of _cdecl and _pascal

Functions declared using **_cdecl** or **_pascal** cannot have the **_fastcall** modifier because they use the stack to pass parameters. Likewise, the **__fastcall** modifier cannot be used together with _export.

The compiler prefixes the **__fastcall** function name with an at-sign ("@"). This prefix applies to both unmangled C function names and to mangled C++ function names.

For Microsoft C++ style **__fastcall** implementation, see __msfastcall and __msreturn.

Note: Note: The __fastcall modifier is subject to name mangling. See the description of the -VC option.

3.1.3.2.18 _stdcall, __stdcall

Category

Modifiers, Keyword extensions

Syntax

```
__stdcall <function-name>
_stdcall <function-name>
```

Description

The **_stdcall** and **__stdcall** keywords force the compiler to generate function calls using the Standard calling convention. Functions must pass the correct number and type of arguments; this is unlike normal C use, which permits a variable number of function arguments.

Such functions comply with the standard WIN32 argument-passing convention.

Note: Note: The __stdcall modifier is subject to name mangling. See the description of the -VC option.

3.1.3.2.19 alignof

Category

Operators, Keyword extensions

Syntax

```
alignof(type);
```

Description

The **alignof** operator tells you the alignment of a type. This feature is one of the C++0x features added to C++Builder 2009.

The result is an integer constant of type `std::size_t`. The value indicates the boundaries on which elements of that type are aligned in memory. An alignment of 2 means that the type must begin on even memory addresses, for instance. A typical value for `alignof(double)` might be 8.

Applying **alignof** to a reference type yields the alignment of the referenced type. If you apply **alignof** to an array type, you get the alignment of its element's type.

See Also

C++0x Features (C++Builder 2009) (see page 462)

Working Draft - Standard for Programming Language C++ (Sec. 5.3.6)

alignof Operator (C++0x) (see page 463)

3.1.3.2.20 and, &&

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **and** operator is an alternative representation of the **&&** operator (binary or logical AND).

If both operands have a value of true, the result has the value true. Otherwise, the result has the value false. Both operands are implicitly converted to bool and the result type is bool.

Unlike the **&** (bitwise AND) operator, the **&&** operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to 0 (or false), the right operand is not evaluated.

Both the **and** and **&&** operators short-circuit (that is, do not evaluate the second operand if the first evaluates as false). Thus, it is both valid and safe to test for a null pointer, as in the following example:

In order to use the **and** operator, you need to check the **Enable new operator names** option (the **-Vn** compiler switch, available on the **Compatibility** page of the **Project** ▸ **Options** dialog box).

Example of and

```
bool test( int * p ) {
  return (p != 0) and (*p > 5);
};
}
```

Substituting **&&** for **and** is also valid.

Overloading && or and

You can overload the && operator as well as the alternative representation, **and**. Keep in mind that when the operator is overloaded, the logic is not short-circuited, and the empty pointer test is not safe. Here is an example demonstrating how to overload **and**:

```
struct example {   bool operator and ( const example & ) {return true;}// same as operator &&}
```

Using && in rvalue References

As part of the new C++0x standard, C++Builder now supports rvalue references. An rvalue reference is formed by placing an && (but not an **and**) after a type in a declaration, as follows:

```
Apt a;Apt&& a_ref1 = a;
```

A major difference between the familiar lvalue reference and the new rvalue reference is that rvalue references can bind to a temporary (that is, an rvalue), but an lvalue reference (at least one that is not a const) cannot bind to an rvalue.

3.1.3.2.21 asm, _asm, __asm

Category

Keyword extensions, C++-Specific Keywords

Syntax

```
asm <opcode> <operands> <; or newline>_asm__asm
_asm <opcode> <operands> <; or newline>
__asm <opcode> <operands> <; or newline>
```

Description

Use the **asm, _asm**, or **__asm** keyword to place assembly language statements in the middle of your C or C++ source code. Any C++ symbols are replaced by the appropriate assembly language equivalents.

You can group assembly language statements by beginning the block of statements with the **asm** keyword, then surrounding the statements with braces ({}).

3.1.3.2.22 auto

Category

Storage class specifiers

Syntax

```
[auto] <data-definition> ;
```

Description

Use the **auto** modifer to define a local variable as having a local lifetime.

This is the default for local variables and is rarely used.

3.1.3.2.23 bitand, &

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **bitand** operator is an alternative representation of the **&**operator (bitwise AND).

The bitwise AND operator compares each bit of the first operand to the corresponding bit of the second operand When both bits are 1, the corresponding result bit is set to 1. Otherwise, the corresponding result bit is set to 0.

In order to use the **bitand** operator, you need to check the **Enable new operator names** option (the **-Vn** compiler switch, available on the **Compatibility** page of the **Project**▸**Options** dialog box).

3.1.3.2.24 bitor, |

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **bitor** operator is an alternative representation of the **|~**operator (bitwise OR).

bitor takes integer or enumeration operands. **bitor** may pass two or more arguments. These arguments must be character or numeric.

When coded as a keyword, **bitor** can only pass two arguments. In either case, the result type is the same as the arguments: (1) unsigned if all arguments are unsigned, or (2) an integer.

In order to use the **bitor** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project**▸**Options** dialog box).

3.1.3.2.25 bool, false, true

Category

C++-Specific Keywords

Syntax

```
bool <identifier>;
```

Description

Use **bool** and the literals **false** and **true** to perform Boolean logic tests.

The **bool** keyword represents a type that can take only the value **false** or **true**. The keywords **false** and **true** are Boolean literals with predefined values. **false** is numericallly zero and **true** is numerically one. These Boolean literals are rvalues; you cannot make an assignment to them.

You can convert an rvalue that is of type **bool** to an rvalue that is **int** type. The numerical conversion sets **false** to zero and **true** becomes one.

You can convert arithmetic, enumeration, pointer, or pointer to member rvalue types to an rvalue of type **bool**. A zero value, null pointer value, or null member pointer value is converted to **false**. Any other value is converted to **true**.

3.1.3.2.26 break

Category

Statements

Syntax

Description

Use the **break** statement within loops to pass control to the first statement following the innermost **switch**, **for**, **while**, or **do** block.

Example

This example illustrates the use of keywords **break**, **case**, **default**, **return**, and **switch**.

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  char ch;

  cout << "PRESS a, b, OR c. ANY OTHER CHOICE WILL TERMINATE THIS PROGRAM." << endl;
  for ( /* FOREVER */; cin >> ch; )
    switch (ch)
    {
      case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
        cout << endl << "Option a was selected." << endl;
        break;
      case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
      case 'c' :
        cout << endl << "Option b or c was selected." << endl;
        break;
      default :
        cout << endl << "NOT A VALID CHOICE!  Bye ..." << endl;
        return(-1);
    }
}
```

See Also

continue (see page 540)

do (see page 546)

for (see page 552)

while (see page 574)

case (see page 535)

default (see page 545)

switch (see page 567)

3.1.3.2.27 case

Category

Statements

Syntax

```
switch ( <switch variable> ){casebreakdefault
case <constant expression> : <statement>; [break;]
    .
    .
    .
default : <statement>;
}
```

Description

Use the **case** statement in conjunction with switches to determine which statements evaluate.

The list of possible branch points within <statement> is determined by preceding substatements with

```
case <constant expression> : <statement>;
```

where <constant expression> must be an **int** and must be unique.

The <constant expression> values are searched for a match for the <switch variable>.

If a match is found, execution continues after the matching **case** statement until a **break** statement is encountered or the end of the **switch** statement is reached.

If no match is found, control is passed to the **default** case.

Note: It is illegal to have duplicate case

constants in the same **switch** statement. **Example**

This example illustrates the use of keywords **break**, **case**, **default**, **return**, and **switch**.

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  char ch;

  cout << "PRESS a, b, OR c. ANY OTHER CHOICE WILL TERMINATE THIS PROGRAM." << endl;
  for ( /* FOREVER */; cin >> ch; )
    switch (ch)
    {
      case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
        cout << endl << "Option a was selected." << endl;
        break;
```

```
        case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
        case 'c' :
          cout << endl << "Option b or c was selected." << endl;
          break;
        default :
          cout << endl << "NOT A VALID CHOICE!  Bye ..." << endl;
          return(-1);
      }
  }
```

See Also

break (see page 534)

default (see page 545)

switch (see page 567)

3.1.3.2.28 catch

Category

Statements, C++-Specific Keywords

Syntax

```
catch (exception-declaration) compound-statement
```

Description

The exception handler is indicated by the **catch** keyword. The handler must be used immediately after the statements marked by the try keyword. The keyword **catch** can also occur immediately after another **catch**. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list.

3.1.3.2.29 cdecl, _cdecl, __cdecl

Category

Modifiers, Keyword extensions

Syntax

```
cdecl <data/function definition> ;_cdecl__cdecl
_cdecl <data/function definition> ;
__cdecl <data/function definition> ;
```

Description

Use a **cdecl**, **_cdecl**, or **__cdecl** modifier to declare a variable or a function using the C-style naming conventions (case-sensitive, with a leading underscore appended). When you use **cdecl**, **_cdecl**, or **__cdecl** in front of a function, it affects how the parameters are passed (parameters are pushed right to left, and the caller cleans up the stack). The **__cdecl** modifier overrides the compiler directives and IDE options.

The **cdecl**, **_cdecl**, and **__cdecl** keywords are specific to CodeGear C++.

3.1.3.2.30 char

Category

Type specifiers

Syntax

```
[signed|unsigned] char <variable_name>
```

Description

Use the type specifier **char** to define a character data type. Variables of type **char** are 1 byte in length.

A **char** can be signed, unsigned, or unspecified. By default, **signed char** is assumed.

Objects declared as characters (**char**) are large enough to store any member of the basic ASCII character set.

3.1.3.2.31 char16_t

Category

C++-Specific Keywords, Type specifiers

Syntax

```
char16_t <identifier>;
```

Description

In C++ programs, **char16_t** is a fundamental data type that can represent a 16 bit character type. This type could be used for UTF-16 characters. You can create a **char16_t** type with `u'<character>'`, which is a literal for a single **char16_t** character

3.1.3.2.32 char32_t

Category

C++-Specific Keywords, Type specifiers

Syntax

```
char32_t <identifier>;
```

Description

In C++ programs, **char32_t** is a fundamental data type that can represent a 32 bit character type. This type could be used for UTF-32 characters. You can create a **char32_t** type with `U'<character>'`, which is a literal for a single **char32_t** character

3.1.3.2.33 class

Category

C++-Specific Keywords, Type specifiers

Syntax

```
<classkey> <classname> <baselist> { <member list> }
```

- <classkey> is either a class**,** struct, or union.
- <classname> can be any name unique within its scope.
- <baselist> lists the base class(es) that this class derives from. <baselist> is optional
- <member list> declares the class's data members and member functions.

Description

Use the **class** keyword to define a C++ class.

Within a class:

- the data are called data members
- the functions are called member functions

See Also

Private (see page 560)

Protected (see page 560)

public (see page 561)

3.1.3.2.34 compl, ~

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **compl** operator is an alternative representation of the **~**operator (bitwise NOT).

The **compl** operator is the abbreviation for complement.

compl Inverts true to false and false to true, The tilde (~) is placed in front of the integer used for the complement.

The complement of 1 would be 0, and vice versa.

Besides an integer, ~ also outputs an enumeration type; this can be the result of the one's complement of the operand. Integral calculations are performed.

The type of the result is the type of the promoted operand.

There can be ambiguity in the unary expression ~X(), where X is a class-name. The ambiguity is resolved in favor of treating ~ as a unary complement rather than treating ~X as a destructor.

In order to use the **compl** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project**▸**Options** dialog box).

3.1.3.2.35 const

Category

Modifiers

Syntax

```
const <variable name> [ = <value> ];
<function name> ( const <type>*<variable name> ;)
<function name> const;
```

Description

Use the **const** modifier to make a variable value unmodifiable.

Use the **const** modifier to assign an initial value to a variable that cannot be changed by the program. Any future assignments to a **const** result in a compiler error.

A **const** pointer cannot be modified, though the object to which it points can be changed. Consider the following examples.

```
const float pi   = 3.14;

// When used by itself, const is equivalent to int.
const  maxint  = 12345;

// A constant pointer
char *const str1 = "Hello, world";

// A pointer to a constant character string.
char const *str2 = "CodeGear Corporation";
```

Given these declarations, the following statements are illegal.

```
pi   = 3.0;         // Assigns a value to a const.
i     = maxint++;   // Increments a const.
str1 = "Hi, there!" // Points str1 to something else.
```

Using the const Keyword in C++ Programs

C++ extends **const** to include classes and member functions. In a C++ class definition, use the **const** modifier following a member function declaration. The member function is prevented from modifying any data in the class.

A class object defined with the **const** keyword attempts to use only member functions that are also defined with **const**. If you call a member function that is not defined as **const**, the compiler issues a warning that a non-const function is being called for a **const** object. Using the **const** keyword in this manner is a safety feature of C++.

Warning: A pointer can indirectly modify a **const** variable, as in the following:

```
*(int *)&my_age = 35;
```

If you use the **const** modifier with a pointer parameter in a function's parameter list, the function cannot modify the variable that the pointer points to. For example,

```
int printf (const char *format, ...);
```

printf is prevented from modifying the format string.

See Also

mutable (see page 555)

3.1.3.2.36 const_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
const_cast< T > (arg)
```

Description

Use the **const_cast** operator to add or remove the **const** or **volatile** modifier from a type.

In the statement, const_cast< T > (arg), T and arg must be of the same type except for **const** and **volatile** modifiers. The cast is resolved at compile time. The result is of type T. Any number of const or volatile modifiers can be added or removed with a single **const_cast** expression.

A pointer to **const** can be converted to a pointer to non-**const** that is in all other respects an identical type. If successful, the resulting pointer refers to the original object.

A **const** object or a reference to **const** cast results in a non-**const** object or reference that is otherwise an identical type.

The **const_cast** operator performs similar typecasts on the **volatile** modifier. A pointer to volatile object can be cast to a pointer

to non-**volatile** object without otherwise changing the type of the object. The result is a pointer to the original object. A **volatile**-type object or a reference to **volatile**-type can be converted into an identical non-**volatile** type.

See Also

Dynamic_cast (typecast Operator) (see page 547)

Reinterpret_cast (typecast Operator) (see page 561)

3.1.3.2.37 continue

Category

Statements

Syntax

```
continue;
```

Description

Use the **continue** statement within loops to pass control to the end of the innermost enclosing end brace belonging to a looping construct, such as **for** or **while**; at which point the loop continuation condition is re-evaluated.

Example

This example illustrates the use of the keyword **continue**.

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  float array[20];

  // Code to initialize array...
  for (int i = 0; i < 20; i++)
  {
    array[i] = random(i + 20);
  }
  array[6] = 0;

        for (int i = 0; i < 20; i++)
  {
    if (array[i] == 0)
      continue;
    array[i] = 1/array[i];
    ListBox1->Items->Add(array[i]);
  }
}
```

See Also

for (see page 552)

while (see page 574)

3.1.3.2.38 __declspec(dllexport)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllexport ) declarator
```

The

dllexport storage-class attribute is used for Microsoft C and C++ language compatibility. This attribute enables you to export functions, data, and objects from a DLL. This attribute explicitly defines the DLL's interface to its client, which can be the executable file or another DLL. Declaring functions as **dllexport** eliminates the need for a module-definition (.DEF) file, at least with respect to the specification of exported functions.

Note: dllexport

replaces the __**export** keyword.

3.1.3.2.39 __declspec(dllimport)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( dllimport ) declarator
```

The **dllimport** storage-class attribute is used for Microsoft C and C++ language compatability. This attribute enables you to import functions, data, and objects to a DLL

Note: Note

: **dllimport** replaces the __**import** keyword.

3.1.3.2.40 __declspec(naked)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( naked ) declarator
```

Use of the naked argument suppresses the prolog/epilog code. Be aware when using __**declspec(naked)** that it does not set up a normal stack frame. A function with __**declspec(naked)** will not preserve the register values that are normally preserved. It is the programmer's responsibility to conform to whatever conventions the caller of the function expects.

You can use this feature to write your own prolog/epilog code using inline assembler code. Naked functions are particularly useful in writing virtual device drivers.

The **naked** attribute is relevant only to the definition of a function and is not a type modifier.

Example

This code defines a function with the **naked** attribute:

```
// Example of the naked attribute
__declspec( naked ) int func( formal_parameters )
{
// Function body
}
```

3.1.3.2.41 __declspec(noreturn)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( noreturn ) declarator
```

This **__declspec** attribute tells the compiler that a function does not return. As a consequence, the compiler knows that the code following a call to a **__declspec(noreturn)** function is unreachable.

If the compiler finds a function with a control path that does not return a value, it generates a warning. If the control path cannot be reached due to a function that never returns, you can use **__declspec(noreturn)** to prevent this warning or error.

Example

Consider the following code. The **else** clause does not contain a return statement, so the programmer declares fatal as **__declspec(noreturn)** to avoid an error or warning message.

```
__declspec(noreturn) extern void fatal ()
{
   // Code omitted
}
int foo()
{
   if(...)
      return 1;
   else if(...)
      return 0;
   else
      fatal();
}
```

3.1.3.2.42 __declspec(nothrow)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( nothrow ) declarator
```

This is a **__declspec** extended attribute that can be used in the declaration of functions. This attribute tells the compiler that the declared function and the functions it calls never throw an exception. With the synchronous exception handling model, now the default, the compiler can eliminate the mechanics of tracking the lifetime of certain unwindable objects in such a function, and significantly reduce the code size.

The following three declarations are equivalent:

```
#define WINAPI __declspec(nothrow) __stdcall
void WINAPI foo1();
void __declspec(nothrow) __stdcall foo2();
void __stdcall foo3() throw();
```

Using **void __declspec(nothrow) __stdcall foo2();** has the advantage that you can use an API definition, such as the illustrated by the **#define** statement, to easily specify nothrow on a set of functions. The third declaration, **void __stdcall foo3() throw();** is the syntax defined by the C++ standard.

3.1.3.2.43 __declspec(novtable)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( novtable ) declarator
```

This form of **_declspec** can be applied to any class declaration, but should only be applied to pure interface classes, that is classes that will never be instantiated on their own. The **_declspec** stops the compiler from generating code to initialize the vfptr in the constructor(s) and destructor of the class. In many cases, this removes the only references to the vtable that are associated with the class and, thus, the linker will remove it. Using this form of **_declspec** can result in a significant reduction in code size

3.1.3.2.44 __declspec(property)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( property( get=get_func_name ) ) declarator
__declspec( property( put=put_func_name ) ) declarator
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

This attribute can be applied to non-static “virtual data members” in a class or structure definition. The compiler treats these “virtual data members” as data members by changing their references into function calls.

When the compiler sees a data member declared with this attribute on the right of a member-selection operator (“**.**” or “**->**“), it converts the operation to a **get** or **put** function, depending on whether such an expression is an l-value or an r-value. In more complicated contexts, such as “**+=**“, a rewrite is performed by doing both **get** and **put**.

This attribute can also be used in the declaration of an empty array in a class or structure definition.

Example

__**declspec**(property(get=GetX, put=PutX)) int x[];

The above statement indicates that x[] can be used with one or more array indices. In this case:

```
i=p->x[a][b]
```

will be turned into:

```
i=p->GetX(a, b),
```

and

```
p->x[a][b] = i
```

will be turned into

```
p->PutX(a, b, i);
```

3.1.3.2.45 __declspec(selectany)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( selectany ) declarator
```

A global data item can normally be initialized only once in an application or library. This attribute can be used in initializing global data defined by headers, when the same header appears in more than one source file.

Note This attribute can only be applied to the actual initialization of global data items that are externally visible.

Example

This code shows how to use the **selectany** attribute:

```
//Correct - x1 is initialized and externally visible
__declspec(selectany) int x1=1;
//Incorrect - const is by default static in C++, so
//x2 is not visible externally (This is OK in C, since
//const is not by default static in C)
const __declspec(selectany) int x2 =2;
//Correct - x3 is extern const, so externally visible
extern const __declspec(selectany) int x3=3;
//Correct - x4 is extern const, so it is externally visible
extern const int x4;
const __declspec(selectany) int x4=4;
//Incorrect - __declspec(selectany) is applied to the uninitialized //declaration of x5 extern __declspec(selectany) int x5;
```

3.1.3.2.46 __declspec(thread)

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( thread ) declarator
```

Thread Local Storage (TLS) is the method by which each thread in a given multithreaded process allocates storage for thread-specific data.

The **thread** extended storage-class modifier is used to declare a thread local variable. The **thread** attribute must be used with the **__declspec** keyword.

3.1.3.2.47 __declspec(uuid(“ComObjectGUID”))

Category

Modifiers, Keyword extensions, Storage class specifiers

Syntax

```
__declspec( uuid(“ComObjectGUID”) ) declarator
```

The compiler attaches a GUID to a class or structure declared or defined (full COM object definitions only) with the **uuid** attribute. The **uuid** attribute takes a string as its argument. This string names a GUID in normal registry format with or without the **{ }** delimiters. For example:

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}")) IDispatch;
```

This attribute can be applied in a redeclaration. This allows the system headers to supply the definitions of interfaces such as

IUnknown, and the redeclaration in some other header (such as COMDEF.H) to supply the GUID.

The keyword __uuidof can be applied to retrieve the constant GUID attached to a user-defined type.

3.1.3.2.48 decltype

Category

Type specifiers

Syntax

```
decltype ( expression )
```

Description

Use the **decltype** type specifier to obtain the type specifier of an expression.

decltype may be applied to expressions that are identifiers, class members and function names or calls.

3.1.3.2.49 default

Category

Statements

Syntax

```
switch ( <switch variable> ){casebreakdefault
case <constant expression> : <statement>; [break;]
    .
    .
    .
default : <statement>;
}
```

Description

Use the default statement in switch statement blocks.

- If a case match is not found and the **default** statement is found within the switch statement, the execution continues at this point.
- If no default is defined in the **switch** statement, control passes to the next statement that follows the switch statement block.

Example

This example illustrates the use of keywords **break**, **case**, **default**, **return**, and **switch**.

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  char ch;

  cout << "PRESS a, b, OR c. ANY OTHER CHOICE WILL TERMINATE THIS PROGRAM." << endl;
  for ( /* FOREVER */; cin >> ch; )
    switch (ch)
    {
      case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
        cout << endl << "Option a was selected." << endl;
        break;
      case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
```

```
        case 'c' :
          cout << endl << "Option b or c was selected." << endl;
          break;
        default :
          cout << endl << "NOT A VALID CHOICE!  Bye ..." << endl;
          return(-1);
      }
}
```

See Also

break (see page 534)

case (see page 535)

switch (see page 567)

3.1.3.2.50 delete

Category

Operators, C++-Specific Keywords

Syntax

```
void operator delete(void *ptr) throw();
void operator delete(void *ptr, const std::nothrow_t&) throw();
void operator delete[](void *ptr) throw();
void operator delete[](void *ptr, const std::nothrow_t &) throw();
void operator delete(void *ptr, void *) throw();   // Placement form
void operator delete[](void *ptr, void *) throw(); // Placement form
```

Description

The **delete** operator deallocates a memory block allocated by a previous call to new. It is similar but superior to the standard library function free.

You should use the **delete** operator to remove all memory that has been allocated by the **new** operator. Failure to free memory can result in memory leaks.

The default placement forms of **operator delete** are reserved and cannot be redefined. The default placement **delete** operator performs no action (since no memory was allocated by the default placement **new** operator). If you overload the placement version of operator new, it is a good idea (though not strictly required) to provide the overload the placement delete operator with the corresponding signature.

See Also

Overloading The Operator delete (see page 436)

The Delete Operator With Arrays (see page 437)

3.1.3.2.51 do

Category

Statements

Syntax

```
do <statement> while ( <condition> );
```

Description

The **do** statement executes until the condition becomes **false**.

<statement> is executed repeatedly as long as the value of <condition> remains **true**.

Since the conditon tests after each the loop executes the <statement>, the loop will execute at least once.

See Also

while (see page 574)

3.1.3.2.52 double

Category

Type specifiers

Syntax

```
[long] double <identifier>
```

Description

Use the **double** type specifier to define an identifier to be a floating-point data type. The optional modifier **long** extends the accuracy of the floating-point value.

If you use the **double** keyword, the floating-point math package will automatically be linked into your program.

See Also

float (see page 552)

long (see page 555)

3.1.3.2.53 dynamic_cast (typecast Operator)

Category

C++-Specific Keywords

Description

In the expression, dynamic_cast< T > (ptr), T must be a pointer or a reference to a defined class type or **void***. The argument ptr must be an expression that resolves to a pointer or reference.

If T is void* then ptr must also be a pointer. In this case, the resulting pointer can access any element of the class that is the most derived element in the hierarchy. Such a class cannot be a base for any other class.

Conversions from a derived class to a base class, or from one derived class to another, are as follows: if T is a pointer and ptr is a pointer to a non-base class that is an element of a class hierarchy, the result is a pointer to the unique subclass. References are treated similarly. If T is a reference and ptr is a reference to a non-base class, the result is a reference to the unique subclass.

A conversion from a base class to a derived class can be performed only if the base is a polymorphic type.

The conversion to a base class is resolved at compile time. A conversion from a base class to a derived class, or a conversion across a hierarchy is resolved at runtime.

If successful, dynamic_cast< T > (ptr) converts ptr to the desired type. If a pointer cast fails, the returned pointer is valued 0. If a cast to a reference type fails, the Bad_cast exception is thrown.

Note: Runtime type identification (RTTI) is required for dynamic_cast

.

See Also

Const_cast (typecast Operator) (see page 539)

Reinterpret_cast (typecast Operator) (see page 561)

__rtti (see page 528)

static_cast (typecast Operator) (see page 565)

3.1.3.2.54 enum

Category

Type specifiers

Syntax

```
enum [<type_tag>] {<constant_name> [= <value>], ...} [var_list];
```

- <type_tag> is an optional type tag that names the set.
- <constant_name> is the name of a constant that can optionally be assigned the value of <value>. These are also called enumeration constants.
- <value> must be an integer. If <value> is missing, it is assumed to be: <prev> + 1 where <prev> is the value of the previous integer constant in the list. For the first integer constant in the list, the default value is 0.
- <var_list> is an optional variable list that assigns variables to the enum type.

Description

Use the **enum** keyword to define a set of constants of type **int**, called an enumeration data type.

An enumeration data type provides mnemonic identifiers for a set of integer values. Use the **-b** flag to toggle the Treat Enums As Ints option. .Enums are always interpreted as **int**s if the range of values permits this, but if they are not **int**s the value gets promoted to an **int** in expressions. Depending on the values of the enumerators, identifiers in an enumerator list are implicitly of type **signed char**, **unsigned char**, **short**, **unsigned short**, **int**, or **unsigned int**.

In C, an enumerated variable can be assigned any value of type int--no type checking beyond that is enforced. In C++, an enumerated variable can be assigned only one of its enumerators.

In C++, you can omit the **enum** keyword if <tag_type> is not the name of anything else in the same scope. You can also omit <tag_type> if no further variables of this **enum** type are required.

In the absence of a <value> the first enumerator is assigned the value of zero. Any subsequent names without initializers will then increase by one. <value> can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers.

In C++, enumerators declared within a class are in the scope of that class.

3.1.3.2.55 explicit

Category

C++-Specific Keywords

Syntax

```
explicit <single-parameter constructor declaration>
```

Description

Normally, a class with a single-parameter constructor can be assigned a value that matches the constructor type. This value is automatically (implicitly) converted into an object of the class type to which it is being assigned. You can prevent this kind of implicit conversion from occurring by declaring the constructor of the class with the **explicit** keyword. Then all objects of that class must be assigned values that are of the class type; all other assignments result in a compiler error.

Objects of the following class can be assigned values that match the constructor type or the class type:

```
class X {
public:
X(int);
X(const char*, int = 0);
};
```

Then, the following assignment statements are legal.

```
void f(X arg) {
X a = 1;
X B = "Jessie";
a = 2;
}
```

However, objects of the following class can be assigned values that match the class type only:

```
class X {
public:
explicit X(int);
explicit X(const char*, int = 0);
};
```

The **explicit** constructors then require the values in the following assignment statements to be converted to the class type to which they are being assigned.

```
void f(X arg) {
X a = X(1);
X b = X("Jessie",0);
a = X(2);
}
```

3.1.3.2.56 export

Category

Unimplemented

Syntax

```
export template < template-parameter-list > declaration
```

Description

The **export** keyword is reserved for future implementation, but has no effect in this release.

See Also

_export (see page 530)

3.1.3.2.57 extern

Category

Storage class specifiers

Syntax

```
extern <data definition> ;
```

```
[extern] <function prototype> ;
```

Description

Use the **extern** modifier to indicate that the actual storage and initial value of a variable, or body of a function, is defined in a separate source code module. Functions declared with extern are visible throughout all source files in a program, unless you redefine the function as static.

The keyword **extern** is optional for a function prototype.

Use extern "C" to prevent function names from being mangled in C++ programs.

Also, **extern** templates allow you to define templates that are not instantiated in a translation unit. Using **extern** templates thus reduces both compilation time and the size of the compiled module. The extern template (see page 494) feature is part of the new C++0x standard.

3.1.3.2.58 final

Category

Attributes, Keyword extensions

Syntax

```
return-type function-name [[final]]
```

Description

The **final** attribute prevents a class or function from being further inherited. You can add the **final** attribute to a class definition or to a virtual member function declaration inside a class definition. If the attribute is specified for a class definition, it is equivalent to being specified for each virtual member function of that class, including inherited member functions.

See Also

Attributes noreturn and final (see page 492)

3.1.3.2.59 __finally

Category

Statements, Keyword extensions

Syntax

```
__finally {compound-statement}
```

Description

The **__finally** keyword specifies actions that should be taken regardless of how the flow within the preceding __try exits.

The following is the code fragment shows how to use the **try/__finally** construct:

```
#include <stdio.h>#include <string.h>#include <windows.h>class
Exception{public:Exception(char* s = "Unknown"){what = strdup(s);      }Exception(const
Exception& e ){what = strdup(e.what); } ~Exception()                    {free(what);         }
char* msg() const             {return what;           }private:char* what;};int main(){float
e, f, g;try  {try    {f = 1.0;g = 0.0;try      {puts("Another exception:");e = f / g;
}__except(EXCEPTION_EXECUTE_HANDLER)      {puts("Caught a C-based
exception.");throw(Exception("Hardware error: Divide by 0"));      }    }catch(const
Exception& e)    {printf("Caught C++ Exception: %s :\n", e.msg());    }  }__finally
{puts("C++ allows __finally too!");  }return e;}
#include <string.h>
#include <windows.h>
```

```
class Exception
{
public:
Exception(char* s = "Unknown"){what = strdup(s);      }
Exception(const Exception& e ){what = strdup(e.what); }
 ~Exception()                    {free(what);         }
  char* msg() const              {return what;          }
private:
char* what;
};
int main()
{
float e, f, g;
try
  {
try
    {
f = 1.0;
g = 0.0;
try
      {
puts("Another exception:");
e = f / g;
      }
__except(EXCEPTION_EXECUTE_HANDLER)
      {
puts("Caught a C-based exception.");
throw(Exception("Hardware error: Divide by 0"));
      }
    }
catch(const Exception& e)
    {
printf("Caught C++ Exception: %s :\n", e.msg());
    }
  }
__finally
  {
puts("C++ allows __finally too!");
  }
return e;
}
#include <iostream>#include <stdexcept>using namespace std;class MyException : public
exception {public:virtual const char *what() const throw() {return("MyException
occurred.");}};// Give me any integer...void myFunc(int a){ MyException e;   // ...but not
that one.   if(a == 0)     throw(e);}void main(){   int g;   // Note __finally must be in its
own try block (with no preceding catch).   try {     try {       g = 0;       myFunc(g);
}     catch(MyException &e) {       cout << "Exception: " << e.what() << endl;     }   }
__finally {       cout << "Finally block reached." << endl;   }
#include <stdexcept>
using namespace std;
class MyException : public exception {
public:
virtual const char *what() const throw() {
return("MyException occurred.");
}
};
// Give me any integer...
void myFunc(int a)
{
 MyException e;
   // ...but not that one.
   if(a == 0)
     throw(e);
}
void main()
{
```

```
    int g;
    // Note __finally must be in its own try block (with no preceding catch).
    try {
      try {
        g = 0;
        myFunc(g);
      }
      catch(MyException &e) {
        cout << "Exception: " << e.what() << endl;
      }
    }
    __finally {
        cout << "Finally block reached." << endl;
    }
}
```

Running the above program results in the following:

```
Another exception:Caught a C-based exception.Caught C++ exception[Hardware error: Divide by
0]C++ allows __finally too!Exception: MyException occurred.Finally block reached.
Caught a C-based exception.Caught C++ exception[Hardware error: Divide by 0]C++ allows
__finally too!Exception: MyException occurred.Finally block reached.
```

3.1.3.2.60 float

Category

Type specifiers

Syntax

```
float <identifier>
```

Description

Use the **float** type specifier to define an identifier to be a floating-point data type.

Type	Length	Range
float	32 bits	3.4 * (10**-38) to 3.4 * (10**+38)

The floating-point math package will be automatically linked into your program if you use floating-point values or operators.

See Also

double (see page 547)

3.1.3.2.61 for

Category

Statements

Syntax

```
for ( [<initialization>] ; [<condition>] ; [<increment>] )  <statement>
```

Description

The **for** statement implements an iterative loop.

<condition> is checked before the first entry into the block.

<statement> is executed repeatedly UNTIL the value of <condition> is false.

- Before the first iteration of the loop, <initialization> initializes variables for the loop.
- After each iteration of the loop, <increments> increments a loop counter. Consequently, j++ is functionally the same as ++j.

In C++, <initialization> can be an expression or a declaration.

The scope of any identifier declared within the **for** loop extends to the end of the control statement only.

A variable defined in the **for**-initialization expression is in scope only within the **for**-block. See the description of the -Vd option.

All the expressions are optional. If <condition> is left out, it is assumed to be always true.

3.1.3.2.62 friend

Category

C++-Specific Keywords

Syntax

```
friend <identifier>;
```

Description

Use **friend** to declare a function or class with full access rights to the private and protected members of the class, without being a member of that class. The outside class has fill access to the class that declares that outside class a friend.

In all other respects, the **friend** is a normal function in terms of scope, declarations, and definitions.

3.1.3.2.63 goto

Category

Statements

Syntax

```
goto <identifier> ;
```

Description

Use the **goto** statement to transfer control to the location of a local label specified by <identifier>.

Labels are always terminated by a colon.

3.1.3.2.64 if, else

Category

Operators

C++ Syntax

```
if ( <condition1> )  <statement1>
if ( <condition1> )  <statement1>;
 else  <statement2>;
if ( <condition1> )  <statement1>;
 else if ( <condition2> )  <statement2>;
 else  <statement3>;
if ( <condition1> )
{
  if ( <condition2> ) {
  <statement1>
  <statement2>
```

```
  }
  else <statement3>
}
else
  <statement4>
```

Description

Use **if** to implement a conditional statement.

You can declare variables in the condition expression. For example,

```
if (int val = func(arg))
```

is valid syntax. The variable val is in scope for the **if** statement and extends to an **else** block when it exists.

The condition statement must convert to a **bool** type. Otherwise, the condition is ill-formed.

When <condition> evaluates to **true**, <statement1> executes.

If <condition> is **false**, <statement2> executes.

The **else** keyword is optional, but no statements can come between an **if** statement and an **else**.

The #if and #else preprocessor statements (directives) look similar to the **if** and **else** statements, but have very different effects. They control which source file lines are compiled and which are ignored.

3.1.3.2.65 import, _import, __import

Category

Modifiers, Keyword extensions

Form 1

```
class _import <class name>
class __import <class name>
```

Form 2

```
return_type _import <function name>
return_type __import <function name>
```

Form 3

```
data_type _import <data name>
data_type __import <data name>
```

Description

This keyword can be used as a class, function, or data modifier.

3.1.3.2.66 inline

Category

C++-Specific Keywords

Syntax

```
inline <datatype> <class>_<function> (<parameters>) { <statements>; }
```

Description

Use the **inline** keyword to declare or define C++ inline functions.

Inline functions are best reserved for small, frequently used functions.

3.1.3.2.67 int

Category

Type specifiers

Syntax

```
[signed|unsigned] int <identifier> ;
```

Description

Use the **int** type specifier to define an integer data type.

Variables of type **int** can be signed (default) or unsigned.

3.1.3.2.68 long

Category

Type specifiers

Syntax

```
long [int] <identifier> ;
[long] double <identifier> ;
```

Description

When used to modify a **double**, it defines a floating-point data type with 80 bits of precision instead of 64.

The floating-point math package will be automatically linked with your program if you use floating-point values or operators.

3.1.3.2.69 mutable

Category

C++-Specific Keywords, Storage class specifiers

Syntax

```
mutable <variable name>;
```

Description

Use the **mutable** specifier to make a variable modifiable even though it is in a **const**-qualified expression.

Using the mutable Keyword

Only class data members can be declared mutable. The **mutable** keyword cannot be used on **static** or **const** names. The purpose of **mutable** is to specify which data members can be modified by **const** member functions. Normally, a **const** member function cannot modify data members.

See Also

const (see page 538)

3.1.3.2.70 namespace

Category

C++-Specific Keywords

Description

Most real-world applications consist of more than one source file. The files can be authored and maintained by more than one developer. Eventually, the separate files are organized and linked to produce the final application. Traditionally, the file organization requires that all names that aren't encapsulated within a defined namespace (such as function or class body, or translation unit) must share the same global namespace. Therefore, multiple definitions of names discovered while linking separate modules require some way to distinguish each name. The solution to the problem of name clashes in the global scope is provided by the C++ namespace mechanism.

The namespace mechanism allows an application to be partitioned into a number of subsystems. Each subsystem can define and operate within its own scope. Each developer is free to introduce whatever identifiers are convenient within a subsystem without worrying about whether such identifiers are being used by someone else. The subsystem scope is known throughout the application by a unique identifier.

It only takes two steps to use C++ namespaces. The first is to uniquely identify a namespace with the keyword **namespace**. The second is to access the elements of an identified namespace by applying the **using** keyword.

3.1.3.2.71 new

Category

Operators, C++-Specific Keywords

Syntax

```
void *operator new(std::size_t size) throw(std::bad_alloc);
void *operator new(std::size_t size, const std::nothrow_t &) throw();
void *operator new[](std::size_t size) throw(std::bad_alloc);
void *operator new[](std::size_t size, const std::nothrow_t &) throw();
void *operator new(std::size_t size, void *ptr) throw();    // Placement form
void *operator new[](std::size_t size, void *ptr) throw();  // Placement form
```

Description

The **new** operators offer dynamic storage allocation, similar but superior to the standard library function malloc. These allocation functions attempt to allocate size bytes of storage. If successful, **new** returns a pointer to the allocated memory. If the allocation fails, the **new** operator will call the new_handler function. The default behavior of new_handler is to throw an exception of type bad_alloc. If you do not want an exception to be thrown, use the nothrow version of **operator new**. The nothrow versions return a null pointer result on failure, instead of throwing an exception.

The default placement forms of **operator new** are reserved and cannot be redefined. You can, however, overload the placement form with a different signature (i.e. one having a different number, or different type of arguments). The default placement forms accept a pointer of type void, and perform no action other than to return that pointer, unchanged. This can be useful when you want to allocate an object at a known address. Using the placement form of **new** can be tricky, as you must remember to explicitly call the destructor for your object, and then free the pre-allocated memory buffer. Do not call the **delete** operator on an object allocated with the placement **new** operator.

A request for non-array allocation uses the appropriate **operator new()** function. Any request for array allocation will call the appropriate **operator new[]()** function. Remember to use the array form of **operator delete[]()**, when deallocating an array created with **operator new[]()**.

Note: Arrays of classes require that a default constructor be defined in the class.

A request for allocation of 0 bytes returns a non-null pointer. Repeated requests for zero-size allocations return distinct, non-null pointers.

Example of Operator new with Nothrow

```
#include <new>
int main(int argc, char* argv[])
{
   int *pn;
        // nothrow version returns null pointer rather than throwing a
        // bad_alloc exception.
        pn = new(nothrow) int[5000000];
        if(pn != NULL) {
          // Allocation succeded.
        }
        return 0;
}
```

See Also

Delete (see page 546)

The Operator new With Arrays (see page 437)

Operator new (see page 437)

Operator new Placement Syntax (see page 436)

Handling Errors For The New Operator (see page 435)

3.1.3.2.72 noreturn

Category

Attributes, Keyword extensions

Syntax

```
void function-name [[noreturn]]
```

Description

Use the **noreturn** modifier to declare functions that have no return value.

See Also

Attributes noreturn and final (see page 492)

3.1.3.2.73 not, !

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **not** operator is an alternative representation of the **!**operator (logical negation).

not returns *true* if its operand is *false*, and *false* if its operand is *true*t.

If an integer is 1, the expression of !1 indicates that the number is other than 1 So that is a true statement. **not** can also be used

for strings as well. The expression !Smith indicates that the person's name is other than Smith.. **not** inverts the bits of the expression.

In order to use the **not** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project**▸**Options** dialog box).

3.1.3.2.74 not_eq, !=

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **not_eq** operator is an alternative representation of **!=** (bitwise inequality). It tests for logical inequivalence.

not_eq compares two expressions to determine whether or not they are the same.

Therefore, 7 != 8 returns true, while 7 != 7 returns false. The same is true for any expression such as Smith != Smith. which returns false.

In order to use the **not_eq** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project**▸**Options** dialog box).

3.1.3.2.75 nullptr

Category

Reserved Words

Syntax

```
nullptr
```

Description

Reserved for future use. **nullptr** is a literal that can be assigned to any pointer to indicate the pointer doesn't point to anything.

Warning: C++Builder does not implement nullptr

but does treat it as a keyword when the -Ax compiler flag is set. Do not use **nullptr** as an identifier.

3.1.3.2.76 operator

Category

Operators, C++-Specific Keywords

Syntax

```
operator <operator symbol>( <parameters> )
{
<statements>;
}
```

Description

Use the **operator** keyword to define a new (overloaded) action of the given operator. When the operator is overloaded as a member function, only one argument is allowed, as *this is implicitly the first argument.

When you overload an operator as a friend, you can specify two arguments.

See Also

class (see page 537)

3.1.3.2.77 or, ||

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **or** operator is an alternative representation of the **||** operator (logical OR).

Only two statements can be evaluated at a time.

or returns true if two values are different, such as 2 or 6.

or returns false if both values are the same, such as 10 and 10.

In order to use the **or** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project**▸**Options** dialog box).

3.1.3.2.78 or_eq, |=

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **or_eq** operator is an alternative representation of **!=**(bitwise inclusive OR).

or_eq tests for logical equivalence or bitwise equality.

or_eq operates on two values. When the first value or the second value is 1, true is returned. When the values are 0, false is returned.

In order to use the **not_eq** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project**▸**Options** dialog box).

3.1.3.2.79 pascal, _pascal, __pascal

Category

Modifiers, Keyword extensions

Syntax

```
pascal <data-definition/function-definition> ;
_pascal <data-definition/function-definition> ;
__pascal <data-definition/function-definition> ;
```

Description

Use the **pascal**, **_pascal**, and **__pascal** keywords to declare a variable or a function using a Pascal-style naming convention

(the name is in uppercase).

In addition, **pascal** declares Delphi language-style parameter-passing conventions when applied to a function header (parameters pushed left to right; the called function cleans up the stack).

In C++ programs, functions declared with the **pascal** modifier are still mangled.

3.1.3.2.80 private

Category

C++-Specific Keywords

Syntax

```
private: <declarations>
```

Description

Access to **private** class members is restricted to member functions within the class, and to **friend** classes.

Class members are **private** by default.

Structure (**struct**) and **union** members are **public** by default. You can override the default access specifier for structures, but not for unions.

Friend declarations can be placed anywhere in the class declaration; friends are not affected by access control specifiers.

See Also

class (see page 537)

friend (see page 553)

Protected (see page 560)

3.1.3.2.81 protected

Category

C++-Specific Keywords

Syntax

```
protected: <declarations>
```

Description

Access to **protected** class members is restricted to member functions within the class, member functions of derived classes, and to **friend** classes.

Structure (**struct**) and **union** members are **public** by default. You can override the default access specifier for structures, but not for unions.

Friend declarations can be placed anywhere in the class declaration; friends are not affected by access control specifiers.

See Also

class (see page 537)

friend (see page 553)

Private (see page 560)

3.1.3.2.82 public

Category

C++-Specific Keywords

Syntax

```
public: <declarations>
```

Description

A **public** class member can be accessed by any function.

Members of a struct or union are **public** by default.

You can override the default access specifier for structures, but not for unions.

Friend declarations can be placed anywhere in the class declaration; friends are not affected by access control specifiers.

See Also

class (see page 537)

friend (see page 553)

Private (see page 560)

Protected (see page 560)

__published (see page 528)

struct (see page 566)

3.1.3.2.83 register

Category

Storage class specifiers

Syntax

```
register <data definition> ;
```

Description

Use the register storage class specifier to store the variable being declared in a CPU register (if possible), to optimize access and reduce code.

Note: The compiler can ignore requests for register allocation. Register allocation is based on the compiler's analysis of how a variable is used.

3.1.3.2.84 reinterpret_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
reinterpret_cast< T > (arg)
```

Description

In the statement, reinterpret_cast< T > (arg), T must be a pointer, reference, arithmetic type, pointer to function, or pointer to member.

A pointer can be explicitly converted to an integral type.

An integral arg can be converted to a pointer. Converting a pointer to an integral type and back to the same pointer type results in the original value.

A yet undefined class can be used in a pointer or reference conversion.

A pointer to a function can be explicitly converted to a pointer to an object type provided the object pointer type has enough bits to hold the function pointer. A pointer to an object type can be explicitly converted to a pointer to a function only if the function pointer type is large enough to hold the object pointer.

See Also

Const_cast (typecast Operator) (see page 539)

Dynamic_cast (typecast Operator) (see page 547)

3.1.3.2.85 return

Category

Statements

Syntax

```
return [ <expression> ] ;
```

Description

Use the **return** statement to exit from the current function back to the calling routine, optionally returning a value.

Example

This example illustrates the use of keywords **break**, **case**, **default**, **return**, and **switch**.

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
int main(int argc, char* argv[])
{
  char ch;

  cout << "PRESS a, b, OR c. ANY OTHER CHOICE WILL TERMINATE THIS PROGRAM." << endl;
  for ( /* FOREVER */; cin >> ch; )
  switch (ch)
  {
    case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
      cout << endl << "Option a was selected." << endl;
      break;
    case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
    case 'c' :
      cout << endl << "Option b or c was selected." << endl;
      break;
    default :
      cout << endl << "NOT A VALID CHOICE!  Bye ..." << endl;
      return(-1);
    }
}
```

3.1.3.2.86 short

Category

Type specifiers

Syntax

```
short int <variable> ;
```

Description

Use the short type modifier when you want a variable smaller than an int. This modifier can be applied to the base type int.

When the base type is omitted from a declaration, int is assumed.

See Also

long (see page 555)

signed (see page 563)

3.1.3.2.87 signed

Category

Type specifiers

Syntax

```
signed <type> <variable> ;
```

Description

Use the signed type modifier when the variable value can be either positive or negative. The signed modifier can be applied to base types **int**, **char**, **long**, **short**, and **__int64**.

When the base type is omitted from a declaration, **int** is assumed.

See Also

char (see page 536)

int (see page 555)

long (see page 555)

short (see page 563)

unsigned (see page 571)

3.1.3.2.88 sizeof

Category

Operators

Description

The **sizeof** operator has two distinct uses:

sizeof unary-expression

sizeof (type-name)

The result in both cases is an integer constant that gives the size in bytes of how much memory space is used by the operand (determined by its type, with some exceptions). The amount of space that is reserved for each type depends on the machine.

In the first use, the type of the operand expression is determined without evaluating the expression (and therefore without side effects). When the operand is of type **char** (**signed** or **unsigned**), **sizeof** gives the result 1. When the operand is a non-parameter of array type, the result is the total number of bytes in the array (in other words, an array name is not converted to a pointer type). The number of elements in an array equals sizeof array/ sizeof array[0] .

If the operand is a parameter declared as array type or function type, **sizeof** gives the size of the pointer. When applied to structures and unions, **sizeof** gives the total number of bytes, including any padding.

You cannot use **sizeof** with expressions of function type, incomplete types, parenthesized names of such types, or with an lvalue that designates a bit field object.

The integer type of the result of **sizeof** is size_t.

You can use **sizeof** in preprocessor directives; this is specific to CodeGear C++.

In C++, sizeof(classtype), where classtype is derived from some base class, returns the size of the object (remember, this includes the size of the base class).

Example

```
/*  USE THE sizeof OPERATOR TO GET SIZES OF DIFFERENT DATA TYPES. */
#include <stdio.h>
struct st {
char *name;
int age;
double height;
};
struct st St_Array[]= {  /* AN ARRAY OF structs */
{ "Jr.",     4,  34.20 },  /* St_Array[0] */
{ "Suzie",  23,  69.75 },  /* St_Array[1] */
};
int main()
{
long double LD_Array[] = { 1.3, 501.09, 0.0007, 90.1, 17.08 };
printf("\nNumber of elements in LD_Array = %d",
sizeof(LD_Array) / sizeof(LD_Array[0]));
/****  THE NUMBER OF ELEMENTS IN THE St_Array. ****/
printf("\nSt_Array has %d elements",
sizeof(St_Array)/sizeof(St_Array[0]));
/****  THE NUMBER OF BYTES IN EACH St_Array ELEMENT.  ****/
printf("\nSt_Array[0] = %d", sizeof(St_Array[0]));
/****  THE TOTAL NUMBER OF BYTES IN St_Array.  ****/
printf("\nSt_Array=%d", sizeof(St_Array));
return 0;
}
```

Output

```
Number of elements in LD_Array = 5
St_Array has 2 elements
St_Array[0] = 16
St_Array= 32
```

3.1.3.2.89 static

Category

Storage class specifiers

Syntax

```
static <data definition> ;static
static <function name> <function definition> ;
```

Description

Use the **static** storage class specifier with a local variable to preserve the last value between successive calls to that function. A **static** variable acts like a local variable but has the lifetime of an external variable.

In a class, data and member functions can be declared **static**. Only one copy of the **static** data exists for all objects of the class.

A **static** member function of a global class has external linkage. A member of a local class has no linkage. A **static** member function is associated only with the class in which it is declared. Therefore, such member functions cannot be **virtual**.

Static member functions can only call other **static** member functions and only have access to **static** data. Such member functions do not have a **this** pointer.

See Also

Static Members (see page 394)

3.1.3.2.90 static_assert

Category

Statements, C++-Specific Keywords

Syntax

```
static_assert (constant-expression, error-message);
```

Description

The **static_assert** keyword is used to test assertions at compile-time, rather than at preprocessor or run-time.

3.1.3.2.91 static_cast (typecast Operator)

Category

C++-Specific Keywords

Syntax

```
static_cast< T > (arg)
```

Description

In the statement, static_cast< T > (arg), T must be a pointer, reference, arithmetic type, or enum type. Both T and arg must be fully known at compile time.

If a complete type can be converted to another type by some conversion method already provided by the language, then making such a conversion by using **static_cast** achieves exactly the same thing.

Integral types can be converted to **enum** types. A request to convert arg to a value that is not an element of **enum** is undefined.

The null pointer is converted to the null pointer value of the destination type, T.

A pointer to one object type can be converted to a pointer to another object type. Note that merely pointing to similar types can cause access problems if the similar types are not similarly aligned.

You can explicitly convert a pointer to a class X to a pointer to some class Y if X is a base class for Y. A static conversion can be made only under the following conditions:

- if an unambiguous conversion exists from Y to X
- if X is not a virtual base class

An object can be explicitly converted to a reference type X& if a pointer to that object can be explicitly converted to an X*. The result of the conversion is an lvalue. No constructors or conversion functions are called as the result of a cast to a reference.

An object or a value can be converted to a class object only if an appropriate constructor or conversion operator has been declared.

A pointer to a member can be explicitly converted into a different pointer-to-member type only if both types are pointers to members of the same class or pointers to members of two classes, one of which is unambiguously derived from the other.

When T is a reference the result of static_cast< T > (arg) is an lvalue. The result of a pointer or reference cast refers to the original expression.

See Also

Const_cast (typecast Operator) (see page 539)

Dynamic_cast (typecast Operator) (see page 547)

Reinterpret_cast (typecast Operator) (see page 561)

3.1.3.2.92 struct

Category

Type specifiers

Syntax

```
struct [<struct type name>] {
[<type> <variable-name[, variable-name, ...]>] ;
    .
    .
    .
} [<structure variables>] ;
```

Description

Use a **struct** to group variables into a single record.

<struct type name> An optional tag name that refers to the structure type.

<structure variables> The data definitions, also optional.

Though both <struct type name> and <structure variables> are optional, one of the two must appear.

You define elements in the record by naming a <type>, followed by one or more <variable-name> (separated by commas).

Separate different variable types by a semicolon.

Use the **.** operator, or the **->** operator to access elements in a structure.

To declare additional variables of the same type, use the keyword **struct** followed by the <struct type name>, followed by the variable names. In C++ the keyword **struct** can be omitted.

Note: The compiler allows the use of anonymous struct embedded within another structure.

See Also

class (see page 537)

public (see page 561)

union (see page 571)

3.1.3.2.93 switch

Category

Statements

Syntax

```
switch ( <switch variable> ) {casebreakdefault
case <constant expression> : <statement>; [break;]
    .
    .
    .
default : <statement>;
}
```

Description

Use the switch statement to pass control to a case that matches the <switch variable>. At which point the statements following the matching case evaluate.

If no case satisfies the condition the default case evaluates.

To avoid evaluating any other cases and relinquish control from the switch, terminate each case with **break**.

Example

This example illustrates the use of keywords **break**, **case**, **default**, **return**, and **switch**.

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
  char ch;

  cout << "PRESS a, b, OR c. ANY OTHER CHOICE WILL TERMINATE THIS PROGRAM." << endl;
  for ( /* FOREVER */; cin >> ch; )
    switch (ch)
    {
      case 'a' :    /* THE CHOICE OF a HAS ITS OWN ACTION. */
        cout << endl << "Option a was selected." << endl;
        break;
      case 'b' :    /* BOTH b AND c GET THE SAME RESULTS. */
      case 'c' :
        cout << endl << "Option b or c was selected." << endl;
        break;
      default :
        cout << endl << "NOT A VALID CHOICE!  Bye ..." << endl;
        return(-1);
    }
}
```

See Also

break (see page 534)

case (see page 535)

default (see page 545)

3.1.3.2.94 template

Category

C++-Specific Keywords

Syntax

```
template-declaration:templateclass
   template < template-argument-list > declaration
template-argument-list:
   template-argument
   template-argument-list, template argument
template-argument:
   type-argument
   argument-declaration
type-argument:
   class typename identifier
template-class-name:
   template-name < template-arg-list >
template-arg-list:
   template-arg
   template-arg-list , template-arg
template-arg:
   expression
   type-name
< template-argument-list > declaration
```

Description

Use templates (also called generics or parameterized types) to construct a family of related functions or classes.

3.1.3.2.95 this

Category

C++-Specific Keywords

Example

```
class X {
  int a;
public:
  X (int b) {this -> a = b;}
};
```

Description

In nonstatic member functions, the keyword **this** is a pointer to the object for which the function is called. All calls to nonstatic member functions pass **this** as a hidden argument.

this is a local variable available in the body of any nonstatic member function. Use it implicitly within the function for member references. It does not need to be declared and it is rarely referred to explicitly in a function definition.

For example, in the call x.func(y) , where y is a member of X, the keyword **this** is set to &x and y is set to this->y, which is equivalent to x.y.

Static member functions do not have a **this** pointer because they are called with no particular object in mind. Thus, a static member function cannot access nonstatic members without explicitly specifying an object with . or ->.

See Also

class (see page 537)

3.1.3.2.96 throw

Category

Statements, C++-Specific Keywords

Syntax

```
throw assignment-expression
```

Description

When an exception occurs, the throw expression initializes a temporary object of the type T (to match the type of argument arg) used in throw(T arg). Other copies can be generated as required by the compiler. Consequently, it can be useful to define a copy constructor for the exception object.

See Also

catch (see page 536)

3.1.3.2.97 try

Category

Statements, C++-Specific Keywords

Syntax

```
try compound-statement handler-list
```

Description

The **try** keyword is supported only in C++ programs. Use __try in C programs. C++ also allows __**try**.

A block of code in which an exception can occur must be prefixed by the keyword **try**. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted. The sequence of steps taken is as follows:

- The program searches for a matching handler
- If a handler is found, the stack is unwound to that point
- Program control is tranferred to the handler

If no handler is found, the program will call the terminate function. If no exceptions are thrown, the program executes in the normal fashion.

See Also

catch (see page 536)

3.1.3.2.98 typedef

Category

Storage class specifiers

Syntax

```
typedef <type definition> <identifier> ;
```

Description

Use the **typedef** keyword to assign the symbol name <identifier> to the data type definition <type definition>.

3.1.3.2.99 typeid

Category

Operators, C++-Specific Keywords

Syntax

```
typeid( expression )
typeid( type-name )
```

Description

You can use **typeid** to get runtime identification of types and expressions. A call to **typeid** returns a reference to an object of type **const** type_info. The returned object represents the type of the **typeid** operand.

If the **typeid** operand is a dereferenced pointer or a reference to a polymorphic type, **typeid** returns the dynamic type of the actual object pointed or referred to. If the operand is non-polymorphic, **typeid** returns an object that represents the static type.

You can use the **typeid** operator with fundamental data types as well as user-defined types.

When the **typeid** operand is a Delphi class object/reference, **typeid** returns the static rather than runtime type.

If the **typeid** operand is a dereferenced NULL pointer, the Bad_typeid exception is thrown.

3.1.3.2.100 typename

Category

C++-Specific Keywords

Syntax 1

```
typename identifier
```

Syntax 2

```
template < typename identifier  > class identifier
```

Description

Use the syntax 1 to reference a type that you have not yet defined. See example 1.

Use syntax 2 in place of the **class** keyword in a template declaration. See example 2.

Note: When using the typename

keyword with templates, the compiler will not always report an error in cases where the ANSI standard requires the typename keyword. The compiler will flag the omission of typename when the compiler is invoked with the **-A** switch. For example, given the following code:

```
#include <stdio.h>
struct A{ typedef int AInt; };
```

Note: The compiler will flag the omission of typename when the compiler is invoked with the -A

switch.

Note: Compile with: bcc32 (no -A switch)

```
bc++bcc32 test.cpp
```

The result is no error. The Compiler should not assume AInt is a **typename**, but it does unless **-A** switch is used

Note: Compile with: bcc32 (-A

switch)

```
bc++bcc32 -A test.cpp
```

The result is:

Error E2089 47071.cpp 7: Identifier 'AInt' cannot have a type qualifier

Error E2303 47071.cpp 7: Type name expected

Error E2139 47071.cpp 7: Declaration missing ;

Both results are as expected.

3.1.3.2.101 union

Category

Type specifiers

Syntax

```
union [<union type name>] {
<type> <variable names> ;
  ...
} [<union variables>] ;
```

Description

Use unions to define variables that share storage space.

The compiler allocates enough storage in a_number to accommodate the largest element in the union.

Unlike a struct, the members of a union occupy the same location in memory. Writing into one overwrites all others.

Use the record selector (.) to access elements of a union .

See Also

class (see page 537)

public (see page 561)

3.1.3.2.102 unsigned

Category

Type specifiers

Syntax

```
unsigned <type> <variable> ;
```

Description

Use the **unsigned** type modifier when variable values will always be positive. The **unsigned** modifer can be applied to base types **int**, **char**, **long, short,** and **__int64**.

When the base type is omitted from a declaration, **int** is assumed.

See Also

char (see page 536)

int (see page 555)

long (see page 555)

short (see page 563)

signed (see page 563)

3.1.3.2.103 using (declaration)

Category

C++-Specific Keywords

Description

You can access namespace members individually with the using-declaration syntax. When you make a using declaration, you add the declared identifier to the local namespace. The grammar is

using-declaration:

using :: unqualified-identifier;

3.1.3.2.104 virtual

Category

C++-Specific Keywords

Syntax

```
virtual class-namevirtual
virtual function-name
```

Description

Use the **virtual** keyword to allow derived classes to provide different versions of a base class function. Once you declare a function as **virtual**, you can redefine it in any derived class, even if the number and type of arguments are the same.

The redefined function overrides the base class function.

3.1.3.2.105 void

Category

Special types

Syntax

```
void identifier
```

Description

void is a special type indicating the absence of any value. Use the **void** keyword as a function return type if the function does not return a value.

```
void hello(char *name)
{
```

```
printf("Hello, %s.",name);
}
```

Use **void** as a function heading if the function does not take any parameters.

```
int init(void)
{
return 1;
}
```

Void Pointers

Generic pointers can also be declared as **void**, meaning that they can point to any type.

void pointers cannot be dereferenced without explicit casting because the compiler cannot determine the size of the pointer object.

3.1.3.2.106 volatile

Category

Modifiers

Syntax

```
volatile <data definition> ;
```

Description

Use the **volatile** modifier to indicate that a background routine, an interrupt routine, or an I/O port can change a variable. Declaring an object to be **volatile** warns the compiler not to make assumptions concerning the value of the object while evaluating expressions in which it occurs because the value could change at any moment. It also prevents the compiler from making the variable a **register** variable.

```
volatile int ticks;
void timer( ) {
ticks++;
}
void wait (int interval) {
ticks = 0;
while (ticks < interval);  // Do nothing
}
```

The routines in this example (assuming timer has been properly associated with a hardware clock interrupt) implement a timed wait of ticks specified by the argument interval. A highly optimizing compiler might not load the value of ticks inside the test of the **while** loop since the loop doesn't change the value of ticks.

Note: C++ extends volatile

to include classes and member functions. If you've declared a **volatile** object, you can use only its **volatile** member functions.

See Also

const (see page 538)

3.1.3.2.107 wchar_t

Category

C++-Specific Keywords, Type specifiers

Syntax

```
wchar_t <identifier>;
```

Description

In C++ programs, **wchar_t** is a fundamental data type that can represent distinct codes for any element of the largest extended character set in any of the supported locales. In CodeGear C++, A **wchar_t** type is the same size, signedness, and alignment requirement as an unsigned short type.

3.1.3.2.108 while

Category

Statements

Syntax

```
while ( <condition> ) <statement>
```

Description

Use the **while** keyword to conditionally iterate a statement.

<statement> executes repeatedly until the value of <condition> is **false**.

The test takes place before <statement> executes. Thus, if <condition> evaluates to **false** on the first pass, the loop does not execute.

3.1.3.2.109 xor, ^

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **xor** operator is an alternative representation of the **^** operator (bitwise xor).

It returns a Boolean true result if just one of its operands is true. This is in opposition to an inclusive **or** which denotes that both statements must be integers for a true statement to be returned.

If 2 or 8.25 are stated to be integers, a true statement will be returned even though 8.25 is a decimal.

If Jack and Jill are both stated to be male, a true statement would be returned even though Jill is female.

In order to use the **xor** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project ▸ Options** dialog box).

3.1.3.2.110 __classmethod

Category

Modifiers, Keyword extensions

Syntax

```
virtual __classmethod int <funcdecl> ( int );
        __classmethod virtual int <funcdecl> (int );
```

Description

The __**classmethod** keyword was added to declare class methods in C++. The modifier __**classmethod** indicates that a function may be invoked on a class name--as well as on an instance of that class.

See Also

__classid (see page 523)

Class Methods (see page 406)

3.1.3.2.111 alignas

Category

C++-Specific Keywords

Syntax

```
alignas
```

Description

Reserved for future use.

Warning: C++Builder does not implement alignas

but does treat it as a keyword when the -Ax compiler flag is set.

3.1.3.2.112 and_eq, &=

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **and_eq** operator is an alternative representation of the **&=** assignment operator (bitwise AND).

The value of the first operand is added to the value of the second operand, and the result is stored in the first operand.

In order to use the **and_eq** operator, you need to check the **Enable new operator names** option (the **-Vn** compiler switch, available on the **Compatibility** page of the **Project** ▸ **Options** dialog box).

See Also

Assignment Operators (see page 591)

3.1.3.2.113 axiom

Category

Reserved Words

Syntax

```
axiom
```

Description

Reserved for future use.

Warning: C++Builder does not implement axiom

but does treat it as a keyword when the -Ax compiler flag is set.

3.1.3.2.114 concept

Category

Reserved Words

Syntax

```
concept
```

Description

Reserved for future use.

Warning: C++Builder does not implement concept

but does treat it as a keyword when the -Ax compiler flag is set.

3.1.3.2.115 concept_map

Category

Reserved Words

Syntax

```
concept_map
```

Description

Reserved for future use.

Warning: C++Builder does not implement concept_map

but does treat it as a keyword when the -Ax compiler flag is set.

3.1.3.2.116 constexpr

Category

Reserved Words

Syntax

```
constexpr
```

Description

Reserved for future use.

Warning: C++Builder does not implement constexpr

but does treat it as a keyword when the -Ax compiler flag is set.

3.1.3.2.117 late_check

Category

Reserved Words

Syntax

```
late_check
```

Description

Reserved for future use.

Warning: C++Builder does not implement late_check

but does treat it as a keyword when the -Ax compiler flag is set.

3.1.3.2.118 requires

Category

Reserved Words

Syntax

```
requires
```

Description

Reserved for future use.

Warning: C++Builder does not implement requires

but does treat it as a keyword when the -Ax compiler flag is set.

3.1.3.2.119 thread_local

Category

Reserved Words

Syntax

```
thread_local
```

Description

Reserved for future use.

Warning: C++Builder does not implement thread_local

but does treat it as a keyword when the -Ax compiler flag is set.

3.1.3.2.120 xor_eq, ^=

Category

Alternative Representations of Operators and Tokens, Operators

Syntax

Description

The **xor_eq** operator is an alternative representation of the **^=;** operator (bitwise XOR assignment)..

True is returned if one number on the left side of an equation is the same as another number on the right side, such as 10*5!=10*2.

False is returned even if 6*3=9*2.

In order to use the **xor** operator, you need to check the **Enable new operator names** option (the **-VM** compiler switch, available on the **Compatibility** page of the **Project**▶**Options** dialog box).

3.1.3.2.121 _Bool

Category

Reserved Words

Syntax

```
_Bool
```

Description

Reserved for future use.

Warning: C++Builder does not implement _Bool

but does treat it as a keyword when the -An compiler flag is set.

3.1.3.2.122 _Complex

Category

Reserved Words

Syntax

```
_Complex
```

Description

Reserved for future use.

Warning: C++Builder does not implement _Complex

but does treat it as a keyword when the -An compiler flag is set.

3.1.3.2.123 _Imaginary

Category

Reserved Words

Syntax

```
_Imaginary
```

Description

Reserved for future use. **_Imaginary** is a literal that can be assigned to any pointer to indicate the pointer doesn't point to anything.

Warning: C++Builder does not implement _Imaginary

but does treat it as a keyword when the -An compiler flag is set. Do not use **_Imaginary** as an identifier.

3.1.3.2.124 restrict

Category

Reserved Words

Syntax

```
restrict
```

Description

Reserved for future use.

Warning: C++Builder does not implement restrict

but does treat it as a keyword when the -An compiler flag is set.

3.1.3.3 Keywords, By Category

This section contains Keywords, By Category topics.

Topics

Name	Description
Alternative Representations of Tokens (see page 579)	This section contains topics about alternative representations of C++ tokens.
Attributes (see page 580)	This section contains C++ attributes keyword topics.
C++Builder Keyword Extensions (see page 580)	This section contains C++Builder Keyword Extension topics.
C++ Specific Keywords (see page 581)	This section contains C++ specific keyword topics.
Modifiers (see page 582)	This section contains C++ Modifier keyword topics.
Operators (see page 583)	This section contains C++ Operator keyword topics.
Reserved Words (see page 584)	This section contains C++ Reserved Words keyword topics.
Special Types (see page 584)	This section contains C++ Special Type keyword topics.
Statement Keywords (see page 584)	This section contains C++ Statement keyword topics.
Storage Class Specifiers (see page 585)	This section contains C++ Storage Class Specifier keyword topics.
Type Specifiers (C++) (see page 586)	This section contains C++ Type Specifier keyword topics.

3.1.3.3.1 Alternative Representations of Tokens

This section contains topics about alternative representations of C++ tokens.

Keyword Topics

The following table summarizes the alternative representation keywords and the token that each keyword represents.

Keyword / Alternative Representation	Token Represented	Definition
and (see page 531)	&&	logical AND
and_eq (see page 575)	&=	assignment by bitwise AND
bitand (see page 533)	&	bitwise AND
bitor (see page 533)	|	bitwise OR
compl (see page 538)	~	bitwise NOT (bitwise one's complement)
not (see page 557)	!	logical negation
not_eq (see page 558)	!=	bitwise inequality

or (see page 559)	||	logical OR
or_eq (see page 559)	|=	bitwise inclusive OR
xor (see page 574)	^	bitwise exclusive OR
xor_eq (see page 577)	^=	bitwise XOR assignment

3.1.3.3.2 Attributes

This section contains C++ attributes keyword topics.

Keyword Topics

final (see page 550)

noreturn (see page 557)

See Also

Attributes noreturn and final (C++0x) (see page 492)

3.1.3.3.3 C++Builder Keyword Extensions

This section contains C++Builder Keyword Extension topics.

Keyword Topics

alignof (see page 531)

asm (see page 532)

__automated (see page 522)

cdecl (see page 536)

__classid (see page 523)

__classmethod (see page 574)

__closure (see page 523)

__declspec (see page 523)

__declspec(dllexport) (see page 540)

__declspec(dllimport) (see page 541)

__declspec(naked) (see page 541)

__declspec(noreturn) (see page 541)

__declspec(nothrow) (see page 542)

__declspec(novtable) (see page 542)

__declspec(property) (see page 543)

__declspec(selectany) (see page 543)

__declspec(thread) (see page 544)

__declspec(uuid(“ComObjectGUID”)) (see page 544)

__except (see page 525)

_export (see page 549)

_fastcall (see page 530)

final (see page 550)

__finally (see page 550)

_import (see page 554)

__inline (see page 525)

__int8 __int16 __int32 __int64 Unsigned_int64 (Extended integer types) (see page 526)

__msfastcall (see page 526)

__msreturn (see page 527)

noreturn (see page 557)

__thread (see page 529)

Pascal (see page 559)

__property (see page 527)

__published (see page 528)

__rtti (see page 528)

_stdcall (see page 531)

__try (see page 529)

3.1.3.3.4 C++ Specific Keywords

This section contains C++ specific keyword topics.

Keyword Topics

asm (see page 532)

bool (see page 533)

catch (see page 536)

char16_t (see page 537)

char32_t (see page 537)

class (see page 537)

const_cast (typecast operator) (see page 529)

delete (see page 546)

dynamic_cast (typecast operator) (see page 547)

explicit (see page 548)

false (see page 533)

friend (see page 553)

inline (see page 554)

mutable (see page 555)

namespace (see page 556)
new (see page 556)
operator (see page 558)
private (see page 560)
protected (see page 560)
public (see page 561)
reinterpret_cast (typecast operator) (see page 561)
__rtti (see page 528)
static_cast (typecst operator) (see page 565)
static_assert (see page 565)
template (see page 568)
this (see page 568)
throw (see page 569)
true (see page 533)
try (see page 569)
typeid (see page 570)
typename (see page 570)
using (declaration) (see page 572)
virtual (see page 572)
whar_t (see page 573)

3.1.3.3.5 Modifiers

This section contains C++ Modifier keyword topics.

Keyword Topics

Cdecl (see page 536)
__classmethod (see page 574)
const (see page 538)
__declspec (see page 523)
__declspec(dllexport) (see page 540)
__declspec(dllimport) (see page 541)
__declspec(naked) (see page 541)
__declspec(noreturn) (see page 541)
__declspec(nothrow) (see page 542)
__declspec(novtable) (see page 542)
__declspec(property) (see page 543)

__declspec(selectany) (see page 543)

__declspec(thread) (see page 544)

__declspec(uuid(“ComObjectGUID”)) (see page 544)

dispid (see page 525)

_export (see page 549)

_fastcall (see page 530)

_import (see page 554)

__msfastcall (see page 526)

__msreturn (see page 527)

Pascal (see page 559)

__rtti (see page 528)

_stdcall (see page 531)

volatile (see page 573)

3.1.3.3.6 Operators

This section contains C++ Operator keyword topics.

Keyword Topics

alignof (see page 531)

and (see page 531)

and_eq (see page 575)

bitand (see page 533)

bitor (see page 533)

__classid (see page 523)

compl (see page 538)

decltype (see page 499)

delete (see page 546)

if (see page 553)

new (see page 556)

not (see page 557)

not_eq (see page 558)

operator (see page 558)

or (see page 559)

or_eq (see page 559)

typeid (see page 570)

sizeof (see page 563)

xor (see page 574)

xor_eq (see page 577)

3.1.3.3.7 Reserved Words

This section contains C++ Reserved Words keyword topics.

Keyword Topic

alignas (see page 575)

axiom (see page 575)

_Bool (see page 578)

_Complex (see page 578)

concept (see page 576)

concept_map (see page 576)

constexpr (see page 576)

_Imaginary (see page 578)

late_check (see page 576)

nullptr (see page 558)

requires (see page 577)

restrict (see page 579)

thread_local (see page 577)

Type Trait Functions (see page 472), such as __is_abstract, and so forth

3.1.3.3.8 Special Types

This section contains C++ Special Type keyword topics.

Keyword Topic

nullptr (see page 558)

void (see page 572)

3.1.3.3.9 Statement Keywords

This section contains C++ Statement keyword topics.

Keyword Topics

break (see page 534)

case (see page 535)

catch (see page 536)

continue (see page 540)

default (see page 545)

do (see page 546)

__except (see page 525)

__finally (see page 550)

for (see page 552)

goto (see page 553)

return (see page 562)

static_assert (see page 565)

switch (see page 567)

throw (see page 569)

__try (see page 529)

try (see page 569)

while (see page 574)

3.1.3.3.10 Storage Class Specifiers

This section contains C++ Storage Class Specifier keyword topics.

Keyword Topics

auto (see page 533)

__declspec (see page 523)

__declspec(dllexport) (see page 540)

__declspec(dllimport) (see page 541)

__declspec(naked) (see page 541)

__declspec(noreturn) (see page 541)

__declspec(nothrow) (see page 542)

__declspec(novtable) (see page 542)

__declspec(property) (see page 543)

__declspec(selectany) (see page 543)

__declspec(thread) (see page 544)

__declspec(uuid(“ComObjectGUID”)) (see page 544)

extern (see page 549)

mutable (see page 555)

register (see page 561)

static (see page 564)

typedef (see page 540)

3.1.3.3.11 Type Specifiers (C++)

This section contains C++ Type Specifier keyword topics.

Keyword Topics

char (see page 536)

case (see page 535)

char16_t (see page 537)

char32_t (see page 537)

class (see page 537)

decltype (see page 545)

double (see page 547)

enum (see page 548)

float (see page 552)

int (see page 555)

long (see page 555)

short (see page 563)

signed (see page 563)

struct (see page 566)

union (see page 571)

unsigned (see page 571)

wchar_t (see page 573)

3.1.3.4 Language Structure

The topics in this section provide a formal definition of C++ language and its implementation in the CodeGear C++ compiler. They describe the legal ways in which tokens can be grouped together to form expressions, statements, and other significant units.

Topics

Name	Description
Binary Operators (see page 587)	This section contains Binary Operator topics.
Declarations (see page 593)	This section briefly reviews concepts related to declarations: objects, storage classes, types, scope, visibility, duration, and linkage. A general knowledge of these is essential before tackling the full declaration syntax. Scope, visibility, duration, and linkage determine those portions of a program that can make legal references to an identifier in order to access its object.
Declaration Syntax (see page 600)	All six interrelated attributes (storage classes, types, scope, visibility, duration, and linkage) are determined in diverse ways by declarations. Declarations can be defining declarations (also known as definitions) or referencing declarations (sometimes known as nondefining declarations). A defining declaration, as the name implies, performs both the duties of declaring and defining; the nondefining declarations require a definition to be added somewhere in the program. A referencing declaration introduces one or more identifier names into a program. A definition actually allocates memory to an object and associates an identifier with that object.

Enumerations (see page 617)	This section contains Enumeration topics.
Expressions (see page 619)	This section contains C++ Expression topics.
Functions (see page 624)	This section contains Function topics.
Operators Summary (see page 635)	This section contains Operator Summary topics.
Pointers (see page 637)	This section contains Pointer topics.
Postfix Expression Operators (see page 642)	This section contains Postfix Expression Operator topics.
Primary Expression Operators (see page 644)	This section contains Primary Expression Operator topics.
Statements (see page 646)	This section contains Statement topics.
Structures (see page 649)	This section contains Structure topics.
Unary Operators (see page 656)	This section contains C++ Unary Operator topics.
Unions (see page 658)	This section contains Union topics.

3.1.3.4.1 Binary Operators

This section contains Binary Operator topics.

Topics

Name	Description
Binary Operators (see page 587)	These are the binary operators in CodeGear C++:
Bitwise Operators (see page 589)	**Syntax**
C++ Specific Operators (see page 590)	The operators specific to C++ are:
Equality Operators (see page 590)	There are two equality operators: **==** and **!=**. They test for equality and inequality between arithmetic or pointer values, following rules very similar to those for the relational operators. **Note:** Notice that == and **!=** have a lower precedence than the relational operators **<** and **>**, **<=**, and **>=**. Also, **==** and **!=** can compare certain pointer types for equality and inequality where the relational operators would not be allowed. The syntax is
Logical Operators (see page 590)	**Syntax**
>, <, >=, <= Relational Operators (see page 591)	**Syntax**
Assignment Operators (see page 591)	**Syntax**
Comma Operator (see page 592)	**Syntax**
Conditional Operators (see page 592)	**Syntax**
Multiplicative Operators (see page 593)	**Syntax**

3.1.3.4.1.1 Binary Operators

These are the binary operators in CodeGear C++:

Arithmetic Operator	Description
+	Binary plus (add)
—	Binary minus (subtract)
*	Multiply
/	Divide
%	Remainder (modulus)

Bitwise Operator	Description
<<	Shift left
>>	Shift right
&	Bitwise AND

^	Bitwise XOR (exclusive OR)
|	Bitwise inclusive OR

Logical Operator	Description
&&	Logical AND

Assignment Operator	Description
=	Assignment
*=	Assign product
/=	Assign quotient
%=	Assign remainder (modulus)
+=	Assign sum
—=	Assign difference
<<=	Assign left shift
>>=	Assign right shift
&=	Assign bitwise AND
^=	Assign bitwise XOR
|=	Assign bitwise OR

Relational Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Component Selection Operator	Description
.	Direct component selector
->	Indirect component selector

Class Member Operator	Description
::	Scope access/resolution
.*	Dereference pointer to class member

->*	Dereference pointer to class member

Conditional Operator	Description
?:	Actually a ternary operator. For example: a ? x : y means "if a then x else y"

Comma Operator	Description
,	Evaluate

3.1.3.4.1.2 Bitwise Operators

Syntax

```
AND-expression &  equality-expression
exclusive-OR-expr ^ AND-expression
inclusive-OR-expr exclusive-OR-expression
~cast-expression
shift-expression << additive-expression
shift-expression >> additive-expression
```

Remarks

Use the bitwise operators to modify the individual bits rather than the number.

Operator	What it does
&	bitwise AND; compares two bits and generates a 1 result if both bits are 1, otherwise it returns 0.
|	bitwise inclusive OR; compares two bits and generates a 1 result if either or both bits are 1, otherwise it returns 0.
^	bitwise exclusive OR; compares two bits and generates a 1 result if the bits are complementary, otherwise it returns 0.
~	bitwise complement; inverts each bit. ~ is used to create destructors.
>>	bitwise shift right; moves the bits to the right, discards the far right bit and if unsigned assigns 0 to the left most bit, otherwise sign extends.
<<	bitwise shift left; moves the bits to the left, it discards the far left bit and assigns 0 to the right most bit.

Both operands in a bitwise expression must be of an integral type.

A	B	A & B	A ^ B	A | B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Note: &, >>, << are context sensitive. & can also be the pointer reference operator.

Note: >> is often overloaded to be the input operator in I/O expressions. << is often overloaded to be the output operator in I/O expressions.

3.1.3.4.1.3 C++ Specific Operators

The operators specific to C++ are:

Operator	Meaning
::	Scope access (or resolution) operator
.*	Dereference pointers to class members
->*	Dereference pointers to pointers to class members
const_cast	adds or removes the **const** or **volatile** modifier from a type
delete	dynamically deallocates memory
dynamic_cast	converts a pointer to a desired type
new	dynamically allocates memory
reinterpret_cast	replaces casts for conversions that are unsafe or implementation dependent
static_cast	converts a pointer to a desired type
typeid	gets run-time identification of types and expressions

Use the scope access (or resolution) operator ::(two semicolons) to access a global (or file duration) name even if it is hidden by a local redeclaration of that name.

Use the .* and ->* operators to dereference pointers to class members and pointers to pointers to class members.

3.1.3.4.1.4 Equality Operators

There are two equality operators: **==** and **!=**. They test for equality and inequality between arithmetic or pointer values, following rules very similar to those for the relational operators.

Note: Notice that ==

and **!=** have a lower precedence than the relational operators **<** and **>**, **<=**, and **>=**. Also, **==** and **!=** can compare certain pointer types for equality and inequality where the relational operators would not be allowed. The syntax is

```
equality-expression:==!=
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression
```

3.1.3.4.1.5 Logical Operators

Syntax

```
logical-AND-expr  && inclusive-OR-expression
logical-OR-expr   || logical-AND-expression
! cast-expression
```

Remarks

Operands in a logical expression must be of scalar type.

&& logical AND; returns **true** only if both expressions evaluate to be nonzero, otherwise returns **false**. If the first expression evaluates to **false**, the second expression is not evaluated.

|| logical OR; returns **true** if either of the expressions evaluate to be nonzero, otherwise returns **false**. If the first expression evaluates to **true**, the second expression is not evaluated.

! logical negation; returns **true** if the entire expression evaluates to be nonzero, otherwise returns **false**. The expression !E is

equivalent to (0 == E).

3.1.3.4.1.6 >, <, >=, <= Relational Operators

Syntax

```
relational-expression  <   shift-expression
relational-expression  >   shift-expression
relational-expression  <=  shift-expression
relational-expression  >=  shift-expression
```

Considerations about Relational Operators

Use relational operators to test equality or inequality of expressions. If the statement evaluates to be **true** it returns a nonzero character; otherwise it returns **false** (0).

Relational Operator	Description
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

In the expression:

```
E1 <operator> E2
```

the operands must follow one of these conditions:

- Both E1 and E2 are of arithmetic type.
- Both E1 and E2 are pointers to qualified or unqualified versions of compatible types.
- Either E1 or E2 is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of void.
- Either E1 or E2 is a pointer, and the other is a null pointer constant.

3.1.3.4.1.7 Assignment Operators

Syntax

```
unary-expr  assignment-op  assignment-expr
```

Remarks

The assignment operators are:

```
=    *=     /=     %=    +=    -=
<<=  >>=    &=     ^=    |=
```

The = operator is the only simple assignment operator, the others are compound assignment operators.

In the expression E1 = E2, E1 must be a modifiable lvalue. The assignment expression itself is not an lvalue.

The expression

```
E1 op= E2
```

has the same effect as

```
E1 = E1 op E2
```

except the lvalue E1 is evaluated only once. For example, E1 += E2 is the same as E1 = E1 + E2.

The expression's value is E1 after the expression evaluates.

For both simple and compound assignment, the operands E1 and E2 must obey one of the following rules:

- 1. E1 is a qualified or unqualified arithmetic type and E2 is an arithmetic type.
- 2. E1 has a qualified or unqualified version of a structure or union type compatible with the type of E2.
- 3. E1 and E2 are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right.
- 4. Either E1 or E2 is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of void. The type pointed to by the left has all the qualifiers of the type pointed to by the right.
- 5. E1 is a pointer and E2 is a null pointer constant.

Note: Spaces separating compound operators (+<space>=) will generate errors.

Note: There are certain conditions where assignment operators are not supported when used with properties.

3.1.3.4.1.8 Comma Operator

Syntax

```
expression , assignment-expression
```

Remarks

The comma separates elements in a function argument list.

The comma is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them.

The left operand E1 is evaluated as a void expression, then E2 is evaluated to give the result and type of the comma expression. By recursion, the expression

```
E1, E2, ..., En
```

results in the left-to-right evaluation of each Ei, with the value and type of En giving the result of the whole expression.

To avoid ambiguity with the commas in function argument and initializer lists, use parentheses. For example,

```
func(i, (j = 1, j + 4), k);
```

calls func with three arguments (i, 5, k), not four.

3.1.3.4.1.9 Conditional Operators

Syntax

```
logical-OR-expr ? expr : conditional-expr
```

Remarks

The conditional operator ?: is a ternary operator.

In the expression E1 ? E2 : E3, E1 evaluates first. If its value is **true**, then E2 evaluates and E3 is ignored. If E1 evaluates to **false**, then E3 evaluates and E2 is ignored.

The result of E1 ? E2 : E3 will be the value of either E2 or E3 depending upon which evaluates.

E1 must be a scalar expression. E2 and E3 must obey one of the following rules:

- 1. Both of arithmetic type. E2 and E3 are subject to the usual arithmetic conversions, which determines the resulting type.
- 2. Both of compatible **struct** or **union** types. The resulting type is the structure or union type of E2 and E3.
- 3. Both of **void** type. The resulting type is **void**.
- 4. Both of type pointer to qualified or unqualified versions of compatible types. The resulting type is a pointer to a type

qualified with all the type qualifiers of the types pointed to by both operands.

- 5. One operand is a pointer, and the other is a null pointer constant. The resulting type is a pointer to a type qualified with all the type qualifiers of the types pointed to by both operands.
- 6. One operand is a pointer to an object or incomplete type, and the other is a pointer to a qualified or unqualified version of **void**. The resulting type is that of the non-pointer-to-**void** operand.

3.1.3.4.1.10 Multiplicative Operators

Syntax

```
multiplicative-expr * cast-expr
multiplicative-expr / cast-expr
multiplicative-expr % cast-expr
```

Remarks

There are three multiplicative operators:

- * (multiplication)
- / (division)
- % (modulus or remainder)

The usual arithmetic conversions are made on the operands.

(op1 * op2) Product of the two operands

(op1 / op2) Quotient of (op1 divided by op2)

(op1 % op2) Remainder of (op1 divided by op2)

For / and %, op2 must be nonzero op2 = 0 results in an error. (You can't divide by zero.)

When op1 and op2 are integers and the quotient is not an integer:

- 1. If op1 and op2 have the same sign, op1 / op2 is the largest integer less than the true quotient, and op1 % op2 has the sign of op1.
- 2. If op1 and op2 have opposite signs, op1 / op2 is the smallest integer greater than the true quotient, and op1 % op2 has the sign of op1.

  Note: Rounding is always toward zero.

  * is context sensitive and can be used as the pointer reference operator.

3.1.3.4.2 Declarations

This section briefly reviews concepts related to declarations: objects, storage classes, types, scope, visibility, duration, and linkage. A general knowledge of these is essential before tackling the full declaration syntax. Scope, visibility, duration, and linkage determine those portions of a program that can make legal references to an identifier in order to access its object.

Topics

Name	Description
Duration (see page 594)	Duration, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike **typedef**s and types, have real memory allocated during run time. There are three kinds of duration: static, local, and dynamic. **Static** Memory is allocated to objects with static duration as soon as execution is underway; this storage allocation lasts until the program terminates. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static... more (see page 594)

Linkage (see page 595)	An executable program is usually created by compiling several independent translation units, then linking the resulting object files with preexisting libraries. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope. Linkage is the process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of three linkage attributes, closely related to their scope: external linkage, internal linkage, or no linkage. These attributes are determined by the placement and format of your declarations,... more (see page 595)
Objects (see page 596)	An object is a specific region of memory that can hold a fixed or variable value (or set of values). (This use of the word object is different from the more general term used in object-oriented languages.) Each value has an associated name and type (also known as a data type). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely references the object. The type is used • to determine the correct memory allocation requirements. • to interpret the bit patterns found in the object during subsequent... more (see page 596)
Scope (see page 597)	The scope of an identifier is that part of the program in which the identifier can be used to access its object. There are six categories of scope: block (or local), function, function prototype, file, class (C++ only), condition (C++ only), and namespace (C++ only). These depend on how and where identifiers are declared. • **Block**. The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the enclosing block). Parameter declarations with a function definition also have block... more (see page 597)
Storage Classes And Types (see page 598)	Associating identifiers with objects requires each identifier to have at least two attributes: storage class and type (sometimes referred to as data type). The C++ compiler deduces these attributes from implicit or explicit declarations in the source code. Storage class dictates the location of the object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the syntax of the declaration, by its placement in the source code, or by both of these factors. The type determines how much memory is allocated to an... more (see page 598)
Translation Units (see page 599)	The term translation unit refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:
Visibility (see page 599)	The visibility of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object. Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended. **Note:** Visibility cannot exceed scope, but scope can exceed visibility. Again, special rules apply to hidden class names and class member names: C++ operators allow hidden identifiers to be... more (see page 599)

3.1.3.4.2.1 Duration

Duration, closely related to storage class, defines the period during which the declared identifiers have real, physical objects allocated in memory. We also distinguish between compile-time and run-time objects. Variables, for instance, unlike **typedef**s and types, have real memory allocated during run time. There are three kinds of duration: static, local, and dynamic.

Static

Memory is allocated to objects with static duration as soon as execution is underway; this storage allocation lasts until the program terminates. All functions, wherever defined, are objects with static duration. All variables with file scope have static duration. Other variables can be given static duration by using the explicit **static** or **extern** storage class specifiers.

Static duration objects are initialized to zero (or null) in the absence of any explicit initializer or, in C++, a class constructor.

Don't confuse static duration with file or global scope. An object can have static duration and local scope.

Local

Local duration objects, also known as automatic objects, lead a more precarious existence. They are created on the stack (or in a register) when the enclosing block or function is entered. They are deallocated when the program exits that block or function. Local duration objects must be explicitly initialized; otherwise, their contents are unpredictable. Local duration objects must always have local or function scope. The storage class specifier **auto** can be used when declaring local duration variables, but is usually redundant, because **auto** is the default for variables declared within a block. An object with local duration also has local scope, because it does not exist outside of its enclosing block. The converse is not true: a local scope object can have static duration.

When declaring variables (for example, **int**, **char**, **float**), the storage class specifier **register** also implies **auto**; but a request (or hint) is passed to the compiler that the object be allocated a register if possible. The compiler can be set to allocate a register to a local integral or pointer variable, if one is free. If no register is free, the variable is allocated as an **auto**, local object with no warning or error.

Note: The compiler can ignore requests for register allocation. Register allocation is based on the compiler's analysis of how a variable is used.

Dynamic

Dynamic duration objects are created and destroyed by specific function calls during a program. They are allocated storage from a special memory reserve known as the heap, using either standard library functions such as malloc, or by using the C++ operator **new**. The corresponding deallocations are made using free or **delete**.

See Also

Declarations (see page 593)

Objects (see page 596)

Storage Classes And Types (see page 598)

Scope (see page 597)

Visibility (see page 599)

Linkage (see page 595)

The keyword static (see page 564)

3.1.3.4.2.2 Linkage

An executable program is usually created by compiling several independent translation units, then linking the resulting object files with preexisting libraries. A problem arises when the same identifier is declared in different scopes (for example, in different files), or declared more than once in the same scope. Linkage is the process that allows each instance of an identifier to be associated correctly with one particular object or function. All identifiers have one of three linkage attributes, closely related to their scope: external linkage, internal linkage, or no linkage. These attributes are determined by the placement and format of your declarations, together with the explicit (or implicit by default) use of the storage class specifier **static** or **extern**.

Each instance of a particular identifier with external linkage represents the same object or function throughout the entire set of files and libraries making up the program. Each instance of a particular identifier with internal linkage represents the same object or function within one file only. Identifiers with no linkage represent unique entities.

External and internal linkage rules

Any object or file identifier having file scope will have internal linkage if its declaration contains the storage class specifier **static**.

For C++, if the same identifier appears with both internal and external linkage within the same file, the identifier will have external

linkage. In C, it will have internal linkage.

If the declaration of an object or function identifier contains the storage class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no such visible declaration, the identifier has external linkage.

If a function is declared without a storage class specifier, its linkage is determined as if the storage class specifier **extern** had been used.

If an object identifier with file scope is declared without a storage class specifier, the identifier has external linkage.

Identifiers with no linkage attribute:

- Any identifier declared to be other than an object or a function (for example, a **typedef** identifier)
- Function parameters
- Block scope identifiers for objects declared without the storage class specifier **extern**

Name mangling

When a C++ module is compiled, the compiler generates function names that include an encoding of the function's argument types. This is known as name mangling. It makes overloaded functions possible, and helps the linker catch errors in calls to functions in other modules. However, there are times when you won't want name mangling. When compiling a C++ module to be linked with a module that does not have mangled names, the C++ compiler has to be told not to mangle the names of the functions from the other module. This situation typically arises when linking with libraries or .obj files compiled with a C compiler

To tell the C++ compiler not to mangle the name of a function, declare the function as extern "C", like this:

```
extern "C" void Cfunc( int );
```

This declaration tells the compiler not to mangle references to the function Cfunc.

You can also apply the extern "C" declaration to a block of names:

```
extern "C" {
   void Cfunc1( int );
   void Cfunc2( int );
   void Cfunc3( int );
};
```

As with the declaration for a single function, this declaration tells the compiler that references to the functions Cfunc1, Cfunc2, and Cfunc3 should not be mangled. You can also use this form of block declaration when the block of function names is contained in a header file:

```
extern "C" {
   #include "locallib.h"
};
```

Note: extern “C” cannot be used with class identifiers.

See Also

Declarations (see page 593)

Objects (see page 596)

Storage Classes And Types (see page 598)

Scope (see page 597)

Visibility (see page 599)

Duration (see page 594)

3.1.3.4.2.3 Objects

An object is a specific region of memory that can hold a fixed or variable value (or set of values). (This use of the word object is

different from the more general term used in object-oriented languages.) Each value has an associated name and type (also known as a data type). The name is used to access the object. This name can be a simple identifier, or it can be a complex expression that uniquely references the object. The type is used

- to determine the correct memory allocation requirements.
- to interpret the bit patterns found in the object during subsequent accesses.
- in many type-checking situations, to ensure that illegal assignments are trapped.

Borland's C++ compiler supports all standard data types, including signed and unsigned integers in various sizes, floating-point numbers in various precisions, structures, unions, arrays, and classes. In addition, pointers to most of these objects can be established and manipulated in memory.

Objects and declarations

Declarations establish the necessary mapping between identifiers and objects. Each declaration associates an identifier with a data type. Most declarations, known as defining declarations, also establish the creation (where and when) of the object; that is, the allocation of physical memory and its possible initialization. Other declarations, known as referencing declarations, simply make their identifiers and types known to the compiler. There can be many referencing declarations for the same identifier, especially in a multifile program, but only one defining declaration for that identifier is allowed.

Generally speaking, an identifier cannot be legally used in a program before its declaration point in the source code. Legal exceptions to this rule (known as forward references) are labels, calls to undeclared functions, and class, struct, or union tags.

lvalues

An lvalue is an object locator: an expression that designates an object. An example of an lvalue expression is *P, where P is any expression evaluating to a non-null pointer. A modifiable lvalue is an identifier or expression that relates to an object that can be accessed and legally changed in memory. A **const** pointer to a constant, for example, is not a modifiable lvalue. A pointer to a constant can be changed (but its dereferenced value cannot).

Historically, the l stood for "left," meaning that an lvalue could legally stand on the left (the receiving end) of an assignment statement. Now only modifiable lvalues can legally stand to the left of an assignment statement. For example, if a and b are nonconstant integer identifiers with properly allocated memory storage, they are both modifiable lvalues, and assignments such as a = 1; and b = a + b are legal.

rvalues

The expression a + b is not an lvalue: a + b = a is illegal because the expression on the left is not related to an object. Such expressions are often called rvalues (short for right values).

See Also

Declarations (see page 593)

Storage Classes And Types (see page 598)

Scope (see page 597)

Visibility (see page 599)

Duration (see page 594)

Linkage (see page 595)

3.1.3.4.2.4 Scope

The scope of an identifier is that part of the program in which the identifier can be used to access its object. There are six categories of scope: block (or local), function, function prototype, file, class (C++ only), condition (C++ only), and namespace (C++ only). These depend on how and where identifiers are declared.

- **Block**. The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such a block is known as the enclosing block). Parameter declarations with a function definition also have block scope, limited to the scope of the block that defines the function.
- **Function**. The only identifiers having function scope are statement labels. Label names can be used with **goto** statements

anywhere in the function in which the label is declared. Labels are declared implicitly by writing label_name: followed by a statement. Label names must be unique within a function.

- **Function prototype**. Identifiers declared within the list of parameter declarations in a function prototype (not part of a function definition) have function prototype scope. This scope ends at the end of the function prototype.
- **File**. File scope identifiers, also known as globals, are declared outside of all blocks and classes; their scope is from the point of declaration to the end of the source file.
- **Class** (C++). A class is a named collection of members, including data structures and functions that act on them. Class scope applies to the names of the members of a particular class. Classes and their objects have many special access and scoping rules; see Classes.
- **Condition** (C++). Declarations in conditions are supported. Variables can be declared within the expression of **if**, **while**, and **switch** statements. The scope of the variable is that of the statement. In the case of an **if** statement, the variable is also in scope for the **else** block.
- **namespace** (C++). A namespace is a logical grouping of program entities (e.g. identifiers, classes, and functions). Namespaces are open, that is, they can span multiple compilation units. You can think of a namespace as introducing a named scope, similar in many ways to a class in C++. See the help for the namespace keyword for more inforrmation on how to declare and use namespaces.

Name spaces

Name space is the scope within which an identifier must be unique. Note that a C++ namespace extends this concept by allowing you to give the scope a name. In addition to the named-scoping capability of C++, the C programming language uses four distinct classes of identifiers:

- **goto** label names. These must be unique within the function in which they are declared.
- Structure, union, and enumeration tags. These must be unique within the block in which they are defined. Tags declared outside of any function must be unique.
- Structure and union member names. These must be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members with the same member name in different structures.
- Variables, **typedef**s, functions, and enumeration members. These must be unique within the scope in which they are defined. Externally declared identifiers must be unique among externally declared variables.

See Also

Declarations (see page 593)

Objects (see page 596)

Storage Classes And Types (see page 598)

Visibility (see page 599)

Duration (see page 594)

Linkage (see page 595)

3.1.3.4.2.5 Storage Classes And Types

Associating identifiers with objects requires each identifier to have at least two attributes: storage class and type (sometimes referred to as data type). The C++ compiler deduces these attributes from implicit or explicit declarations in the source code.

Storage class dictates the location of the object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the syntax of the declaration, by its placement in the source code, or by both of these factors.

The type determines how much memory is allocated to an object and how the program will interpret the bit patterns found in the object's storage allocation. A given data type can be viewed as the set of values (often implementation-dependent) that identifiers of that type can assume, together with the set of operations allowed on those values. The compile-time operator, **sizeof**, lets you determine the size in bytes of any standard or user-defined type. See sizeof for more on this operator.

See Also

Declarations (see page 593)

Objects (see page 596)

Scope (see page 597)

Visibility (see page 599)

Duration (see page 594)

Linkage (see page 595)

3.1.3.4.2.6 Translation Units

The term translation unit refers to a source code file together with any included files, but less any source lines omitted by conditional preprocessor directives. Syntactically, a translation unit is defined as a sequence of external declarations:

```
translation-unit:
external-declaration
translation-unit external-declaration
external-declaration
function-definition
declaration
```

word external has several connotations in C; here it refers to declarations made outside of any function, and which therefore have file scope. (External linkage is a distinct property; see the section Linkage..) Any declaration that also reserves storage for an object or function is called a definition (or defining declaration). For more details, see External declarations and definitions.

See Also

Declarations (see page 593)

Objects (see page 596)

Storage Classes And Types (see page 598)

Scope (see page 597)

Visibility (see page 599)

Duration (see page 594)

Linkage (see page 595)

3.1.3.4.2.7 Visibility

The visibility of an identifier is that region of the program source code from which legal access can be made to the identifier's associated object.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier is ended.

Note: Visibility cannot exceed scope, but scope can exceed visibility.

Again, special rules apply to hidden class names and class member names: C++ operators allow hidden identifiers to be accessed under certain conditions

See Also

Declarations (see page 593)

Objects (see page 596)

Storage Classes And Types (see page 598)

Scope (see page 597)

Duration (see page 594)

Linkage (see page 595)

3.1.3.4.3 Declaration Syntax

All six interrelated attributes (storage classes, types, scope, visibility, duration, and linkage) are determined in diverse ways by declarations.

Declarations can be defining declarations (also known as definitions) or referencing declarations (sometimes known as nondefining declarations). A defining declaration, as the name implies, performs both the duties of declaring and defining; the nondefining declarations require a definition to be added somewhere in the program. A referencing declaration introduces one or more identifier names into a program. A definition actually allocates memory to an object and associates an identifier with that object.

Topics

Name	Description
Tentative Definitions (see page 602)	The ANSI C standard supports the concept of the tentative definition. Any external data declaration that has no storage class specifier and no initializer is considered a tentative definition. If the identifier declared appears in a later definition, then the tentative definition is treated as if the **extern** storage class specifier were present. In other words, the tentative definition becomes a simple referencing declaration. If the end of the translation unit is reached and no definition has appeared with an initializer for the identifier, then the tentative definition becomes a full definition, and the object defined has uninitialized (zero-filled) space... more (see page 602)
Possible Declarations (see page 603)	The range of objects that can be declared includes • Variables • Functions • Classes and class members (C++) • Types • Structure, union, and enumeration tags • Structure members • Union members • Arrays of other types • Enumeration constants • Statement labels • Preprocessor macros The full syntax for declarations is shown in Tables 2.1 through 2.3. The recursive nature of the declarator syntax allows complex declarators. You'll probably want to use **typedefs** to improve legibility. In CodeGear C++ declaration syntax., note the restrictions on the number and order of modifiers and qualifiers. Also, the modifiers listed are the only addition to the declarator syntax that are... more (see page 603)

External Declarations and Definitions (see page 606)	The storage class specifiers auto and register cannot appear in an external declaration. For each identifier in a translation unit declared with internal linkage, no more than one external definition can be given. An external definition is an external declaration that also defines an object or function; that is, it also allocates storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of sizeof), then exactly one external definition of that identifier must exist in the entire program. The C++ compiler allows later declarations of external names, such as arrays,... more (see page 606)
Type Specifiers (see page 607)	The type determines how much memory is allocated to an object and how the program interprets the bit patterns found in the object's storage allocation. A data type is the set of values (often implementation-dependent) identifiers can assume, together with the set of operations allowed on those values. The type specifier with one or more optional modifiers is used to specify the type of the declared identifier:
Type Categories (see page 608)	Provides information on C++ type categories. The four basic type categories (and their subcategories) are as follows:
The Fundamental Types (see page 609)	The fundamental type specifiers are built from the following keywords:
Initialization (see page 611)	Initializers set the initial value that is stored in an object (variables, arrays, structures, and so on). If you don't initialize an object, and it has static duration, it will be initialized by default in the following manner: • To zero if it is an arithmetic type • To null if it is a pointer type **Note:** If the object has automatic storage duration, its value is indeterminate. **Syntax for initializers**
Declaration and Declarators (see page 612)	A **declaration** is a list of names. The names are sometimes referred to as **declarators** or identifiers. The declaration begins with optional storage class specifiers, type specifiers, and other modifiers. The identifiers are separated by commas and the list is terminated by a semicolon. Simple declarations of variable identifiers have the following pattern:
Use of Storage Class Specifiers (see page 614)	Provides information on C++ storage class specifiers. Storage classes specifiers are also called type specifiers. They dictate the location (data segment, register, heap, or stack) of an object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code). Storage class can be established by the declaration syntax, by its placement in the source code, or by both of these factors. The keyword **mutable** does not affect the lifetime of the class member to which it is applied. The storage class specifiers in C++ are: • auto (see page 533) • __declspec (see page 523) • extern (see page 549) • mutable (see page 555) • register (see page 561) • static (see page 564)... more (see page 614)
Variable Modifiers (see page 614)	In addition to the storage class specifier keywords, a declaration can use certain modifiers to alter some aspect of the identifier. The modifiers available are summarized in CodeGear C++ modifiers. The following table summarizes the effects of a modifier applied to a called function. For every modifier, the table shows the order in which the function parameters are pushed on the stack. Next, the table shows whether the calling program (the caller) or the called function (the callee) is responsible for popping the parameters off the stack. Finally, the table shows the effect on the name of a global function.... more (see page 614)

Mixed-Language Calling Conventions (see page 615)	Provides information on C++ mixed-language calling conventions. This section describes C++ mixed-language calling conventions. You can call routines written in other languages, and vice versa. When you mix languages, you have to deal with two important issues: identifiers and parameter passing. By default, the compiler saves all global identifiers in their original case (lower, upper, or mixed) with an underscore "_" prepended to the front of the identifier. To remove the default, you can use the **-u** command-line option. **Note:** The section Linkage (see page 595) tells how to use extern , which allows C names to be referenced from a C++ program.... more (see page 615)
Multithread Variables (see page 616)	The following topic describes C++ multithread variables.
Function Modifiers (see page 616)	This section presents descriptions of the function modifiers available with the CodeGear C++ compiler. You can use the __**declspec(dllexport),** and __**declspec(dllimport)**and __saveregs modifiers to modify functions. In 32-bit programs the keyword can be applied to class, function, and variable declarations The __**declspec(dllexport)** modifier makes the function exportable from Windows. The __**declspec(dllimport)** modifier makes a function available to a Windows program. The keywords are used in an executable (if you don't use smart callbacks) or in a DLL. Functions declared with the __**fastcall** modifier have different names than their non-__**fastcall** counterparts. The compiler prefixes the __**fastcall** function name with... more (see page 616)

3.1.3.4.3.1 Tentative Definitions

The ANSI C standard supports the concept of the tentative definition. Any external data declaration that has no storage class specifier and no initializer is considered a tentative definition. If the identifier declared appears in a later definition, then the tentative definition is treated as if the **extern** storage class specifier were present. In other words, the tentative definition becomes a simple referencing declaration.

If the end of the translation unit is reached and no definition has appeared with an initializer for the identifier, then the tentative definition becomes a full definition, and the object defined has uninitialized (zero-filled) space reserved for it. For example,

```
int x;
int x;          /*legal, one copy of x is reserved */
int y;
int y = 4;      /* legal, y is initialized to 4 */
int z = 5;
int z = 6;      /* not legal, both are initialized definitions */
```

Unlike ANSI C, C++ doesn't have the concept of a tentative declaration; an external data declaration without a storage class specifier is always a definition.

See Also

Declaration Syntax

Possible Declarations (see page 603)

External Declarations And Definitions (see page 606)

Type Categories (see page 608)

The Fundamental Types (see page 609)

Initialization (see page 611)

Declarations And Declarators (see page 612)

Variable Modifiers (see page 614)

Function Modifiers (see page 616)

3.1.3.4.3.2 Possible Declarations

The range of objects that can be declared includes

- Variables
- Functions
- Classes and class members (C++)
- Types
- Structure, union, and enumeration tags
- Structure members
- Union members
- Arrays of other types
- Enumeration constants
- Statement labels
- Preprocessor macros

The full syntax for declarations is shown in Tables 2.1 through 2.3. The recursive nature of the declarator syntax allows complex declarators. You'll probably want to use **typedefs** to improve legibility.

In CodeGear C++ declaration syntax., note the restrictions on the number and order of modifiers and qualifiers. Also, the modifiers listed are the only addition to the declarator syntax that are not ANSI C or C++. These modifiers are each discussed in greater detail in Variable Modifiers and Function Modifiers.

CodeGear C++ declaration syntax

declaration:	elaborated-type-specifier:
<decl-specifiers> <declarator-list>;	class-key identifier
asm-declaration	class-key class-name
function-declaration	**enum** enum-name
linkage-specification	class-key: (C++ specific)
decl-specifier:	**class**
storage-class-specifier	**struct**
type-specifier	**union**
function-specifier	enum-specifier:
friend (C++ specific)	**enum** <identifier> { <enum-list> }
typedef	enum-list:
decl-specifiers:	enumerator
<decl-specifiers> decl-specifier	enumerator-list , enumerator
storage-class-specifier:	enumerator:
auto	identifier
register	identifier = constant-expression
static	constant-expression:
extern	conditional-expression

function-specifier: (C++ specific)	linkage-specification: (C++ specific)
inline	**extern** string { <declaration-list> }
virtual	**extern** string declaration
simple-type-name:	type-specifier:
class-name	simple-type-name
typedef-name	class-specifier
boolean	
char	enum-specifier
short	elaborated-type-specifier
int	**const**
__int8	
__int16	
__int32	
__int64	
long	**volatile**
signed	declaration-list:
unsigned	declaration
float	declaration-list ; declaration
double	
void	
declarator-list:	type-name:
init-declarator	type-specifier <abstract-declarator>
declarator-list , init-declarator	abstract-declarator:
init-declarator:	pointer-operator <abstract-declarator>
declarator <initializer>	<abstract-declarator> (argument-declaration-list)
declarator:	<cv-qualifier-list>
dname	<abstract-declarator> [<constant-expression>]
modifier-list	(abstract-declarator)
pointer-operator declarator	argument-declaration-list:
declarator (parameter-declaration-list)	<arg-declaration-list>
<cv-qualifier-list >	arg-declaration-list , ...
(The <cv-qualifier-list > is for C++ only.)	<arg-declaration-list> **...** (C++ specific)
declarator [<constant-expression>]	arg-declaration-list:
(declarator)	argument-declaration

modifier-list:	arg-declaration-list , argument-declaration
modifier	argument-declaration:
modifier-list modifier	decl-specifiers declarator
modifier:	decl-specifiers declarator = expression
__cdecl	(C++ specific)
__pascal	decl-specifiers <abstract-declarator>
__stdcall	decl-specifiers <abstract-declarator> **=** expression
__fastcall	(C++ specific)
function-definition:	
function-body:	
pointer-operator:	compound-statement
• <cv-qualifier-list>	initializer:
& <cv-qualifier-list> (C++ specific)	**=** expression
class-name :: * <cv-qualifier-list>	= { initializer-list }
(C++ specific)	(expression-list) (C++ specific)
cv-qualifier-list:	initializer-list:
cv-qualifier <cv-qualifier-list>	expression
cv-qualifier	initializer-list , expression
const	{ initializer-list <,> }
volatile	
dname:	
name	
class-name (C++ specific)	
~ class-name (C++ specific)	
type-defined-name	

See Also

Declaration Syntax

Tentative Definitions (see page 602)

External Declarations And Definitions (see page 606)

Type Categories (see page 608)

The Fundamental Types (see page 609)

Initialization (see page 611)

Declarations And Declarators (see page 612)

Variable Modifiers (see page 614)

Function Modifiers (see page 616)

3.1.3.4.3.3 External Declarations and Definitions

The storage class specifiers auto and register cannot appear in an external declaration. For each identifier in a translation unit declared with internal linkage, no more than one external definition can be given.

An external definition is an external declaration that also defines an object or function; that is, it also allocates storage. If an identifier declared with external linkage is used in an expression (other than as part of the operand of sizeof), then exactly one external definition of that identifier must exist in the entire program.

The C++ compiler allows later declarations of external names, such as arrays, structures, and unions, to add information to earlier declarations. Here's an example:

```
int a[];              // no size
struct mystruct;      // tag only, no member declarators
  .
  .
  .
int a[3] = {1, 2, 3}; // supply size and initialize
struct mystruct {
   int i, j;
};                    // add member declarators
```

CodeGear C++ class declaration syntax (C++ only) covers class declaration syntax. In the section on classes (beginning with Classes), you can find examples of how to declare a class. Referencing covers C++ reference types (closely related to pointer types) in detail. Finally, see Using Templates for a discussion of **template**-type classes.

CodeGear C++ class declaration syntax (C++ only)

class-specifier: base-specifier:

class-head { <member-list> } : base-list

class-head: base-list:

class-key <identifier> <base-specifier> base-specifier

class-key class-name <base-specifier> base-list , base-specifier

member-list: base-specifier:

member-declaration <member-list> class-name

access-specifier : <member-list> **virtual** <access-specifier> class-name

member-declaration: access-specifier <**virtual**> class-name

<decl-specifiers> <member-declarator-list> ; access-specifier:

function-definition <;> **private**

qualified-name ; **protected**

member-declarator-list: **public**

member-declarator conversion-function-name:

member-declarator-list, member-declarator **operator** conversion-type-name

member-declarator: conversion-type-name:

declarator <pure-specifier> type-specifiers <pointer-operator>

<identifier> : constant-expression constructor-initializer:

pure-specifier: **:** member-initializer-list

= 0

member-initializer-list: operator-name: one of

member-initializer **new delete sizeof typeid**

member-initializer , member-initializer-list **+ - * / % ^**

member-initializer: **& | ~ ! = <>**

class name (<argument-list>) **+= -= =* /= %= ^=**

identifier (<argument-list>) **&= |= << >> >>= <<=**

operator-function-name: **== != <= >= && ||**

operator operator-name **++ __ , ->* -> ()**

[ ] .*

See Also

Declaration Syntax (see page 600)

Tentative Definitions (see page 602)

Possible Declarations (see page 603)

Type Categories (see page 608)

The Fundamental Types (see page 609)

Initialization (see page 611)

Declarations And Declarators (see page 612)

Variable Modifiers (see page 614)

Function Modifiers (see page 616)

3.1.3.4.3.4 Type Specifiers

The type determines how much memory is allocated to an object and how the program interprets the bit patterns found in the object's storage allocation. A data type is the set of values (often implementation-dependent) identifiers can assume, together with the set of operations allowed on those values.

The type specifier with one or more optional modifiers is used to specify the type of the declared identifier:

```
int i;                  // declare i as an integer
unsigned char ch1, ch2;  // declare two unsigned chars
```

By long-standing tradition, if the type specifier is omitted, type **signed int** (or equivalently, **int**) is the assumed default. However, in C++, a missing type specifier can lead to syntactic ambiguity, so C++ practice requires you to explicitly declare all **int** type specifiers.

The type specifier keywords by the CodeGear C++ compiler are:

char wchar_t	float	signed
class	int	struct
double	long	union

enum	short	unsigned

Use the sizeof operators to find the size in bytes of any predefined or user-defined type.

3.1.3.4.3.5 Type Categories

Provides information on C++ type categories.

The four basic type categories (and their subcategories) are as follows:

```
Aggregate   Array
   struct
   union
   class (C++ only)
Function
Scalar
   Arithmetic
   Enumeration
   Pointer
   Reference (C++ only)
void
```

Types can also be viewed in another way: they can be fundamental or derived types. The fundamental types are **void**, **char**, **int**, **float**, and **double**, together with **short**, **long**, **signed**, and **unsigned** variants of some of these. The derived types include pointers and references to other types, arrays of other types, function types, class types, structures, and unions.

A class object, for example, can hold a number of objects of different types together with functions for manipulating these objects, plus a mechanism to control access and inheritance from other classes.

Given any nonvoid type **type** (with some provisos), you can declare derived types as follows:

Declaring Types

Declaration	Description
type t;	An object of type **type**
type array[10];	Ten **type**s: array[0] - array[9]
type *ptr;	ptr is a pointer to **type**
type &ref = t;	ref is a reference to **type** (C++)
type func(**void**);	func returns value of type **type**
void func1(**type** t);	func1 takes a type **type** parameter
struct st {**type** t1;**type** t2};	structure **st** holds two **type**s

Note: type& var, type &var, and type & var are all equivalent.

See Also

Declaration Syntax

Tentative Definitions (see page 602)

Possible Declarations (see page 603)

External Declarations And Definitions (see page 606)

The Fundamental Types (see page 609)

Initialization (see page 611)

Declarations And Declarators (see page 612)

Variable Modifiers (see page 614)

Function Modifiers (see page 616)

The keyword void (see page 572)

3.1.3.4.3.6 The Fundamental Types

The fundamental type specifiers are built from the following keywords:

char	**__int8**	**long**
double	**__int16**	**signed**
float	**__int32**	**short**
int	**__int64**	**unsigned**

From these keywords you can build the integral and floating-point types, which are together known as the arithmetic types. The modifiers **long**, **short**, **signed**, and **unsigned** can be applied to the integral types. The include file limits.h contains definitions of the value ranges for all the fundamental types.

Integral types

char, **short**, **int**, and **long**, together with their unsigned variants, are all considered integral data types. Integral types shows the integral type specifiers, with synonyms listed on the same line.

Integral types

char, signed char	Synonyms if default **char** set to **signed.**
unsigned char	
char, unsigned char	Synonyms if default **char** set to **unsigned.**
signed char	
int, signed int	
unsigned, unsigned int	
short, short int, signed short int	
unsigned short, unsigned short int	
long, long int, signed long int	
unsigned long, unsigned long int	

Note: These synonyms are not valid in C++. See The three char types.

signed or **unsigned** can only be used with **char**, **short**, **int**, or **long**. The keywords **signed** and **unsigned**, when used on their own, mean **signed int** and **unsigned int**, respectively.

In the absence of **unsigned**, **signed** is assumed for integral types. An exception arises with **char**. You can set the default for **char** to be **signed** or **unsigned**. (The default, if you don't set it yourself, is **signed**.) If the default is set to **unsigned**, then the declaration char ch declares ch as **unsigned**. You would need to use signed char ch to override the default. Similarly, with a **signed** default for **char**, you would need an explicit unsigned char ch to declare an **unsigned char**.

Only **long** or **short** can be used with **int**. The keywords **long** and **short** used on their own mean **long int** and **short int**.

ANSI C does not dictate the sizes or internal representations of these types, except to indicate that **short**, **int**, and **long** form a nondecreasing sequence with **"short <= int <= long."** All three types can legally be the same. This is important if you want to write portable code aimed at other platforms.

The compiler regards the types **int** and **long** as equivalent, both being 32 bits. The signed varieties are all stored in two's complement format using the most significant bit (MSB) as a sign bit: 0 for positive, 1 for negative (which explains the ranges shown in 32-bit data types, sizes, and ranges). In the unsigned versions, all bits are used to give a range of 0 - (2n - 1), where n is 8, 16, or 32.

Floating-point types

The representations and sets of values for the floating-point types are implementation dependent; that is, each implementation of C is free to define them. The compiler uses the IEEE floating-point formats.See the topic on ANSI implementation-specific.

float and **double** are 32- and 64-bit floating-point data types, respectively. **long** can be used with **double** to declare an 80-bit precision floating-point identifier: **long double** test_case, for example.

The table of 32-bit data types, sizes, and ranges indicates the storage allocations for the floating-point types

Standard arithmetic conversions

When you use an arithmetic expression, such as a + b, where a and b are different arithmetic types, The compiler performs certain internal conversions before the expression is evaluated. These standard conversions include promotions of "lower" types to "higher" types in the interests of accuracy and consistency.

Here are the steps the compiler uses to convert the operands in an arithmetic expression:

- 1. Any small integral types are converted as shown in Methods used in standard arithmetic conversions. After this, any two values associated with an operator are either **int** (including the **long** and **unsigned** modifiers), or they are of type **double**, **float**, or **long double**.
- 2. If either operand is of type **long double**, the other operand is converted to **long double**.
- 3. Otherwise, if either operand is of type **double**, the other operand is converted to **double**.
- 4. Otherwise, if either operand is of type **float**, the other operand is converted to **float**.
- 5. Otherwise, if either operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
- 6. Otherwise, if either operand is of type **long**, then the other operand is converted to **long**.
- 7. Otherwise, if either operand is of type **unsigned**, then the other operand is converted to **unsigned**.
- 8. Otherwise, both operands are of type **int**.

The result of the expression is the same type as that of the two operands.

Methods used in standard arithmetic conversions

Type	Converts to	Method
char	**int**	Zero or sign-extended (depends on default char type)
unsigned char	**int**	Zero-filled high byte (always)
signed char	**int**	Sign-extended (always)
short	**int**	Same value; sign extended
unsigned short	**unsigned int**	Same value; zero filled
enum	**int**	Same value

Special char, int, and enum conversions

Note: The conversions discussed in this section are specific to the Borland C++ compiler.

Assigning a signed character object (such as a variable) to an integral object results in automatic sign extension. Objects of type **signed char** always use sign extension; objects of type **unsigned char** always set the high byte to zero when converted to **int**.

Converting a longer integral type to a shorter type truncates the higher order bits and leaves low-order bits unchanged. Converting a shorter integral type to a longer type either sign-extends or zero-fills the extra bits of the new value, depending on whether the shorter type is **signed** or **unsigned**, respectively.

See Also

Declaration Syntax

Tentative Definitions (see page 602)

Possible Declarations (see page 603)

External Declarations And Definitions (see page 606)

Type Categories (see page 608)

Initialization (see page 611)

Declarations And Declarators (see page 612)

Variable Modifiers (see page 614)

Function Modifiers (see page 616)

3.1.3.4.3.7 Initialization

Initializers set the initial value that is stored in an object (variables, arrays, structures, and so on). If you don't initialize an object, and it has static duration, it will be initialized by default in the following manner:

- To zero if it is an arithmetic type
- To null if it is a pointer type

Note: If the object has automatic storage duration, its value is indeterminate.

Syntax for initializers

```
initializer
        = expression
        = {initializer-list} <,>}
        (expression list)
initializer-list
        expression
        initializer-list, expression
        {initializer-list} <,>}
```

Rules governing initializers

The number of initializers in the initializer list cannot be larger than the number of objects to be initialized.

The item to be initialized must be an object (for example, an array).

For C (not required for C++), all expressions must be constants if they appear in one of these places:

In an initializer for an object that has static duration.

In an initializer list for an array, structure, or union (expressions using **sizeof** are also allowed).

- If a declaration for an identifier has block scope, and the identifier has external or internal linkage, the declaration cannot have an initializer for the identifier.
- If a brace-enclosed list has fewer initializers than members of a structure, the remainder of the structure is initialized implicitly in the same way as objects with static storage duration.

Scalar types are initialized with a single expression, which can optionally be enclosed in braces. The initial value of the object is that of the expression; the same constraints for type and conversions apply as for simple assignments.

For unions, a brace-enclosed initializer initializes the member that first appears in the union's declaration list. For structures or unions with automatic storage duration, the initializer must be one of the following:

- An initializer list (as described in Arrays, structures, and unions).
- A single expression with compatible union or structure type. In this case, the initial value of the object is that of the expression.

Arrays, structures, and unions

You initialize arrays and structures (at declaration time, if you like) with a brace-enclosed list of initializers for the members or elements of the object in question. The initializers are given in increasing array subscript or member order. You initialize unions with a brace-enclosed initializer for the first member of the union. For example, you could declare an array days, which counts how many times each day of the week appears in a month (assuming that each day will appear at least once), as follows:

```
int days[7] = { 1, 1, 1, 1, 1, 1, 1 }
```

The following rules initialize character arrays and wide character arrays:

- You can initialize arrays of character type with a literal string, optionally enclosed in braces. Each character in the string, including the null terminator, initializes successive elements in the array. For example, you could declare

*

```
char name[] = { "Unknown" };
```

which sets up an eight-element array, whose elements are 'U' (for name[0]), 'n' (for name[1]), and so on (and including a null terminator).

- You can initialize a wide character array (one that is compatible with wchar_t) by using a wide string literal, optionally enclosed in braces. As with character arrays, the codes of the wide string literal initialize successive elements of the array.

* Here is an example of a structure initialization:

```
struct mystruct {
   int i;
   char str[21];
   double d;
   } s = { 20, "Borland", 3.141 };
```

Complex members of a structure, such as arrays or structures, can be initialized with suitable expressions inside nested braces.

See Also

Declaration Syntax

Tentative Definitions (see page 602)

Possible Declarations (see page 603)

External Declarations And Definitions (see page 606)

Type Categories (see page 608)

The Fundamental Types (see page 609)

Declarations And Declarators (see page 612)

Variable Modifiers (see page 614)

Function Modifiers (see page 616)

3.1.3.4.3.8 Declaration and Declarators

A **declaration** is a list of names. The names are sometimes referred to as **declarators** or identifiers. The declaration begins with optional storage class specifiers, type specifiers, and other modifiers. The identifiers are separated by commas and the list is

terminated by a semicolon.

Simple declarations of variable identifiers have the following pattern:

```
data-type var1 <=init1>, var2 <=init2>, ...;
```

where var1, var2,... are any sequence of distinct identifiers with optional initializers. Each of the variables is declared to be of type data-type. For example:

```
int x = 1, y = 2;
```

creates two integer variables called x and y (and initializes them to the values 1 and 2, respectively).

These are all defining declarations; storage is allocated and any optional initializers are applied.

The initializer for an automatic object can be any legal expression that evaluates to an assignment-compatible value for the type of the variable involved. Initializers for static objects must be constants or constant expressions.

In C++, an initializer for a static object can be any expression involving constants and previously declared variables and functions.

The format of the declarator indicates how the declared name is to be interpreted when used in an expression. If **type** is any type, and storage class specifier is any storage class specifier, and if D1 and D2 are any two declarators, then the declaration:

```
storage-class-specifier type D1, D2;
```

indicates that each occurrence of D1 or D2 in an expression will be treated as an object of type **type** and storage class storage class specifier. The type of the name embedded in the declarator will be some phrase containing **type**, such as "**type**," "pointer to **type**," "array of **type**," "function returning **type**," or "pointer to function returning **type**," and so on.

For example, in Declaration syntax examples each of the declarators could be used as rvalues (or possibly lvalues in some cases) in expressions where a single **int** object would be appropriate. The types of the embedded identifiers are derived from their declarators as follows:

Declaration syntax examples

Declarator syntax	Implied type of name	Example
type	name; type	**int** count;
type name[];	(open) array of **type**	**int** count[];
type name[3];	Fixed array of three elements, **int** count[3] all of **type**;	(name[0], name[1], and name[2]
type *name;	Pointer to **type**	**int** *count;
type *name[];	(open) array of pointers to **type**	**int** *count[];
type *(name[]);	Same as above	**int** *(count[]);
type (*name)[];	Pointer to an (open) array of **type**	**int** (*count) [];
type &name;	Reference to **type** (C++ only)	**int** &count;
type name();	Function returning **type**	**int** count();
type *name();	Function returning pointer to **type**	**int** *count();
type *(name());	Same as above	**int** *(count());
type (*name)();	Pointer to function returning **type**	**int** (*count) ();

Note the need for parentheses in (*name)[ ] and (*name)( ); this is because the precedence of both the array declarator [ ] and the function declarator ( ) is higher than the pointer declarator *. The parentheses in *(name[ ]) are optional.

Note: See CodeGear C++ declaration syntax for the declarator syntax. The definition covers both identifier and function

declarators.

See Also

Declaration Syntax

Tentative Definitions (see page 602)

Possible Declarations (see page 603)

External Declarations And Definitions (see page 606)

Type Categories (see page 608)

The Fundamental Types (see page 609)

Initialization (see page 611)

Variable Modifiers (see page 614)

Function Modifiers (see page 616)

3.1.3.4.3.9 Use of Storage Class Specifiers

Provides information on C++ storage class specifiers.

Storage classes specifiers are also called type specifiers. They dictate the location (data segment, register, heap, or stack) of an object and its duration or lifetime (the entire running time of the program, or during execution of some blocks of code).

Storage class can be established by the declaration syntax, by its placement in the source code, or by both of these factors.

The keyword **mutable** does not affect the lifetime of the class member to which it is applied.

The storage class specifiers in C++ are:

- auto (see page 533)
- __declspec (see page 523)
- extern (see page 549)
- mutable (see page 555)
- register (see page 561)
- static (see page 564)
- typedef (see page 569)

See Also

Storage Class Specifier Keywords (see page 585)

3.1.3.4.3.10 Variable Modifiers

In addition to the storage class specifier keywords, a declaration can use certain modifiers to alter some aspect of the identifier. The modifiers available are summarized in CodeGear C++ modifiers.

The following table summarizes the effects of a modifier applied to a called function. For every modifier, the table shows the order in which the function parameters are pushed on the stack. Next, the table shows whether the calling program (the caller) or the called function (the callee) is responsible for popping the parameters off the stack. Finally, the table shows the effect on the name of a global function.

Calling Conventions

Modifier	Push parameters	Pop parameters	Name change (only in C)
__cdecl1	Right to left	Caller	'_' prepended
__fastcall	Left to right	Callee	'@' prepended
__pascal	Left to right	Callee	Uppercase
__stdcall	Right to left	Callee	No change

1. This is the default.

Note: Note: __fastcall and __stdcall are always name mangled in C++. See the description of the -VC option in BCC32.

See Also

Declaration Syntax (see page 600)

Tentative Definitions (see page 602)

Possible Declarations (see page 603)

External Declarations And Definitions (see page 606)

Type Categories (see page 608)

The Fundamental Types (see page 609)

Initialization (see page 611)

Declarations And Declarators (see page 612)

Function Modifiers (see page 616)

Mixed-Language Calling Conventions (see page 615)

The keyword const (see page 538)

The keyword volatile (see page 573)

3.1.3.4.3.11 Mixed-Language Calling Conventions

Provides information on C++ mixed-language calling conventions.

This section describes C++ mixed-language calling conventions.

You can call routines written in other languages, and vice versa. When you mix languages, you have to deal with two important issues: identifiers and parameter passing.

By default, the compiler saves all global identifiers in their original case (lower, upper, or mixed) with an underscore "_" prepended to the front of the identifier. To remove the default, you can use the **-u** command-line option.

Note: The section Linkage (see page 595) tells how to use extern

, which allows C names to be referenced from a C++ program.

Topics

Cdecl (see page 536)

_fastcall (see page 530)

Pascal (see page 559)

_stdcall (see page 531)

3.1.3.4.3.12 Multithread Variables

The following topic describes C++ multithread variables.

Topics

__thread (see page 529)

3.1.3.4.3.13 Function Modifiers

This section presents descriptions of the function modifiers available with the CodeGear C++ compiler.

You can use the **__declspec(dllexport),** and **__declspec(dllimport)**and **__**saveregs modifiers to modify functions.

In 32-bit programs the keyword can be applied to class, function, and variable declarations

The **__declspec(dllexport)** modifier makes the function exportable from Windows. The **__declspec(dllimport)** modifier makes a function available to a Windows program. The keywords are used in an executable (if you don't use smart callbacks) or in a DLL.

Functions declared with the **__fastcall** modifier have different names than their non-**__fastcall** counterparts. The compiler prefixes the **__fastcall** function name with an @. This prefix applies to both unmangled C function names and to mangled C++ function names.

CodeGear C++ modifiers

Modifier	Use with	Description
const1	Variables	Prevents changes to object.
volatile1	Variables	Prevents register allocation and some optimization. Warns compiler that object might be subject to outside change during evaluation.
__cdecl2	Functions	Forces C argument-passing convention. Affects linker and link-time names.
__cdecl2	Variables	Forces global identifier case-sensitivity and leading underscores in C.
__pascal	Functions	Forces Pascal argument-passing convention. Affects linker and link-time names.
__pascal	Variables	Forces global identifier case-insensitivity with no leading underscores in C.
__import	Functions/classes	Tells the compiler which functions or classes to import.
__export	Functions/classes	Tells the compiler which functions or classes to export.
__declspec(dllimport)	Functions/classes	Tells the compiler which functions or classes to import. This is the preferred method.
__declspec(dllexport)	Functions/classes	Tells the compiler which functions or classes to export. This is the preferred method.
__fastcall	Functions	Forces register parameter passing convention. Affects the linker and link-time names.
__stdcall	Function	Forces the standard WIN32 argument-passing convention.

1. C++ extends **const** and **volatile** to include classes and member functions.
2. This is the default.

See Also

Declaration Syntax

Tentative Definitions (see page 602)

Possible Declarations (see page 603)

External Declarations And Definitions (see page 606)

Type Categories (see page 608)

The Fundamental Types (see page 609)

Initialization (see page 611)

Declarations And Declarators (see page 612)

Variable Modifiers (see page 614)

3.1.3.4.4 Enumerations

This section contains Enumeration topics.

Topics

Name	Description
Enumerations (see page 617)	An enumeration data type is used to provide mnemonic identifiers for a set of integer values. For example, the following declaration:
Assignment To Enum Types (see page 618)	The rules for expressions involving **enum** types have been made stricter. The compiler enforces these rules with error messages if the compiler switch **-A** is turned on (which means strict ANSI C++). Assigning an integer to a variable of **enum** type results in an error:

3.1.3.4.4.1 Enumerations

An enumeration data type is used to provide mnemonic identifiers for a set of integer values. For example, the following declaration:

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

establishes a unique integral type, **enum days**, a variable anyday of this type, and a set of enumerators (sun, mon,...) with constant integer values.

The -b compiler switch controls the “Treat Enums As Ints” option. When this switch is used, the compiler allocates a whole word (a four-byte **int**) for enumeration types (variables of type **enum**). The default is ON (meaning **enums** are always **int**s) if the range of values permits, but the value is always promoted to an **int** when used in expressions. The identifiers used in an enumerator list are implicitly of type **signed char**, **unsigned char**, or **int**, depending on the values of the enumerators. If all values can be represented in a **signed** or **unsigned char**, then that is the type of each enumerator.

In C, a variable of an enumerated type can be assigned any value of type **int**--no type checking beyond that is enforced. In C++, a variable of an enumerated type can be assigned only one of its enumerators. That is:

```
anyday = mon;       // OK
anyday = 1;         // illegal, even though mon == 1
```

The identifier days is the optional enumeration tag that can be used in subsequent declarations of enumeration variables of type **enum days**:

```
enum days payday, holiday; // declare two variables
```

In C++, you can omit the **enum** keyword if **days** is not the name of anything else in the same scope.

As with **struct** and **union** declarations, you can omit the tag if no further variables of this **enum** type are required:

```
enum { sun, mon, tues, wed, thur, fri, sat } anyday;
/* anonymous enum type */
```

The enumerators listed inside the braces are also known as enumeration constants. Each is assigned a fixed integral value. In the absence of explicit initializers, the first enumerator (sun) is set to zero, and each succeeding enumerator is set to one more than its predecessor (mon = 1, tues = 2, and so on). See Enumeration constants for more on enumeration constants.

With explicit integral initializers, you can set one or more enumerators to specific values. Any subsequent names without initializers will then increase by one. For example, in the following declaration:

```
/* Initializer expression can include previously declared enumerators */
enum coins { penny = 1, tuppence, nickel = penny + 4, dime = 10,
             quarter = nickel * nickel } smallchange;
```

tuppence would acquire the value 2, nickel the value 5, and quarter the value 25.

The initializer can be any expression yielding a positive or negative integer value (after possible integer promotions). These values are usually unique, but duplicates are legal.

enum types can appear wherever **int** types are permitted.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
typedef enum days DAYS;
DAYS *daysptr;
int i = tues;
anyday = mon;        // OK
*daysptr = anyday;   // OK
mon = tues;          // ILLEGAL: mon is a constant
```

Enumeration tags share the same name space as structure and union tags. Enumerators share the same name space as ordinary variable identifiers:

```
int mon = 11;
{
   enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
   /* enumerator mon hides outer declaration of int mon */
   struct days { int i, j;};   // ILLEGAL: days duplicate tag
   double sat;                 // ILLEGAL: redefinition of sat
}
mon = 12;                      // back in int mon scope
```

In C++, enumerators declared within a class are in the scope of that class.

In C++ it is possible to overload most operators for an enumeration. However, because the =, [ ], ( ), and -> operators must be overloaded as member functions, it is not possible to overload them for an **enum**. See the example on how to overload the postfix and prefix increment operators.

3.1.3.4.4.2 Assignment To Enum Types

The rules for expressions involving **enum** types have been made stricter. The compiler enforces these rules with error messages if the compiler switch **-A** is turned on (which means strict ANSI C++).

Assigning an integer to a variable of **enum** type results in an error:

```
enum color
{
  red, green, blue
};
int f()
{
  color c;
  c = 0;
```

```
  return c;
}
```

The same applies when passing an integer as a parameter to a function. Notice that the result type of the expression flag1|flag2 is **int**:

```
enum e
{
  flag1 = 0x01,
  flag2 = 0x02
};
void p(e);
void f()
{
  p(flag1|flag2);
}
```

To make the example compile, the expression flag1|flag2 must be cast to the **enum** type: e (flag1|flag2).

3.1.3.4.5 Expressions

This section contains C++ Expression topics.

Topics

Name	Description
Expressions (C++) (see page 620)	An expression is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in CodeGear C++ expressions, indicates that expressions are defined recursively: subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.) **Note:** CodeGear C++ expressions show how identifiers and operators are combined to form grammatically legal "phrases." Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules that depend on the operators used, the presence of parentheses, and the data types of the operands. The standard... more (see page 620)
Precedence Of Operators (see page 622)	There are 16 precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other. Where duplicates of operators appear in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence. The precedence of each operator in the following table is indicated by its order in the table. The first category (on the first line) has the highest precedence. Operators on the... more (see page 622)
Expressions And C++ (see page 623)	C++ allows the overloading of certain standard C operators, as explained in Overloading Operator Functions. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the equality operator == might be defined in class complex to test the equality of two complex numbers without changing its normal usage with non-class data types. An overloaded operator is implemented as a function; this function determines the operand type, lvalue, and evaluation order to be applied when the overloaded operator is used. However, overloading cannot change the precedence of an operator. Similarly, C++... more (see page 623)
Evaluation Order (see page 624)	The order in which the compiler evaluates the operands of an expression is not specified, except where an operator specifically states otherwise. The compiler will try to rearrange the expression in order to improve the quality of the generated code. Care is therefore needed with expressions in which a value is modified more than once. In general, avoid writing expressions that both modify and use the value of the same object. For example, consider the expression
Errors And Overflows (see page 624)	Associativity and precedence of CodeGear C++ operators. summarizes the precedence and associativity of the operators. During the evaluation of an expression, the compiler can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo 2n arithmetic on n-bit registers), but errors detected by math library functions can be handled by standard or user-defined routines.See _matherr and signal.

3.1.3.4.5.1 Expressions (C++)

An expression is a sequence of operators, operands, and punctuators that specifies a computation. The formal syntax, listed in CodeGear C++ expressions, indicates that expressions are defined recursively: subexpressions can be nested without formal limit. (However, the compiler will report an out-of-memory error if it can't compile an expression that is too complex.)

Note: CodeGear C++ expressions show how identifiers and operators are combined to form grammatically legal "phrases."

Expressions are evaluated according to certain conversion, grouping, associativity, and precedence rules that depend on the operators used, the presence of parentheses, and the data types of the operands. The standard conversions are detailed in Methods used in standard arithmetic conversions. The way operands and subexpressions are grouped does not necessarily specify the actual order in which they are evaluated by the compiler (see Evaluation order).

Expressions can produce an lvalue, an rvalue, or no value. Expressions might cause side effects whether they produce a value or not

The precedence and associativity of the operators are summarized in Associativity and precedence of CodeGear C++ operators. The grammar in CodeGear C++ expressions, completely defines the precedence and associativity of the operators.

CodeGear C++ expressions

```
primary-expression:
    literal
    this (C++ specific)
    :: identifier (C++ specific)
    :: operator-function-name (C++ specific)
    :: qualified-name (C++ specific)
    (expression)
    name

teral:
    integer-constant
    character-constant
    floating-constant
    string-literal
name:
    identifier
    operator-function-name (C++ specific)
    conversion-function-name (C++ specific)

~ class-name (C++ specific)
    qualified-name (C++ specific)

qualified-name: (C++ specific)
    qualified-class-name :: name
postfix-expression:
    primary-expression
    postfix-expression  [ expression ]
    postfix-expression  (<expression—list>)
   simple-type-name (<expression-list>) (C++ specific)

postfix-expression  .  name
    postfix-expression  ->  name
    postfix-expression  ++

postfix-expression  --
    const_cast < type-id > ( expression ) (C++ specific)
    dynamic_cast < type-id > ( expression ) (C++ specific)
    reinterpret_cast < type-id > ( expression ) (C++ specific)
    static_cast < type-id > ( expression ) (C++ specific)
    typeid ( expression ) (C++ specific)
    typeid ( type-name ) (C++ specific)
expression-list:
```

```
    assignment-expression
    expression-list  ,  assignment-expression
unary-expression:
    postfix-expression
    ++ unary-expression
    - - unary-expression
    unary-operator  cast-expression
    sizeof unary-expression
    sizeof ( type-name )
    allocation-expression (C++ specific)
    deallocation-expression (C++ specific)
unary-operator: one of & * + - !
allocation-expression: (C++ specific)
    <::> new <placement> new-type-name <initializer>
    <::> new <placement> (type-name) <initializer>
placement: (C++ specific)
   (expression-list)
new-type-name: (C++ specific)
   type-specifiers <new-declarator>
new-declarator: (C++ specific)
   ptr-operator <new-declarator>
   new-declarator [<expression>]
deallocation-expression: (C++ specific)
    <::> delete cast-expression
    <::> delete [] cast-expression
cast-expression:
    unary-expression
    ( type-name )  cast-expression

pm-expression:
    cast-expression
    pm-expression .* cast-expression (C++ specific)
    pm-expression ->* cast-expression (C++ specific)
multiplicative-expression:
    pm-expression
    multiplicative-expression  *  pm-expression
    multiplicative-expression  /  pm-expression
    multiplicative-expression  %  pm-expression
additive-expression:
    multiplicative-expression
    additive-expression  +  multiplicative-expression
    additive-expression  -  multiplicative-expression
shift-expression:
additive-expression
    shift-expression  <<  additive-expression
    shift-expression  >>  additive-expression
relational-expression:
    shift-expression
    relational-expression  <   shift-expression
    relational-expression  >   shift-expression
    relational-expression  <=  shift-expression
    relational-expression  >=  shift-expressionequality-expression:
    relational-expression
    equality expression  ==  relational-expression
    equality expression  !=  relational-expression
AND-expression:
    equality-expression
    AND-expression  &  equality-expression
exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression ^ AND-expression
inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression | exclusive-OR-expression
logical-AND-expression:
```

```
inclusive-OR-expression
    logical-AND-expression && inclusive-OR-expression
logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || logical-AND-expression
conditional-expression:
    logical-OR-expression
    logical-OR-expression ? expression : conditional-expression
assignment-expression:
    conditional-expression
    unary-expression  assignment-operator  assignment-expression
assignment-operator: one of

=   *=  /=  %=  +=  -=

<<    =>   >=   &=  ^=  |=
expression:
    assignment-expression
    expression , assignment-expression
constant-expression:
    conditional-expression
```

3.1.3.4.5.2 Precedence Of Operators

There are 16 precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other.

Where duplicates of operators appear in the table, the first occurrence is unary, the second binary. Each category has an associativity rule: left to right, or right to left. In the absence of parentheses, these rules resolve the grouping of expressions with operators of equal precedence.

The precedence of each operator in the following table is indicated by its order in the table. The first category (on the first line) has the highest precedence. Operators on the same line have equal precedence.

Operators	Associativity
() [] -> :: .	left to right
! ~ + - ++ -- & * sizeof new delete	right to left
.* ->*	left to right

* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	right to left
?:	left to right
= *= /= %= += -= &= ^= |= <<= >>=	right to left
,	left to right

3.1.3.4.5.3 Expressions And C++

C++ allows the overloading of certain standard C operators, as explained in Overloading Operator Functions. An overloaded operator is defined to behave in a special way when applied to expressions of class type. For instance, the equality operator == might be defined in class complex to test the equality of two complex numbers without changing its normal usage with non-class data types.

An overloaded operator is implemented as a function; this function determines the operand type, lvalue, and evaluation order to be applied when the overloaded operator is used. However, overloading cannot change the precedence of an operator. Similarly, C++ allows user-defined conversions between class objects and fundamental types. Keep in mind, then, that some of the C language rules for operators and conversions might not apply to expressions in C++.

3.1.3.4.5.4 Evaluation Order

The order in which the compiler evaluates the operands of an expression is not specified, except where an operator specifically states otherwise. The compiler will try to rearrange the expression in order to improve the quality of the generated code. Care is therefore needed with expressions in which a value is modified more than once. In general, avoid writing expressions that both modify and use the value of the same object. For example, consider the expression

```
i = v[i++];  // i is undefined
```

The value of i depends on whether i is incremented before or after the assignment. Similarly,

```
int total = 0;
sum = (total = 3) + (++total); // sum = 4 or sum = 7 ??
```

is ambiguous for sum and total. The solution is to revamp the expression, using a temporary variable:

```
int temp, total = 0;
temp = ++total;
sum = (total = 3) + temp;
```

Where the syntax does enforce an evaluation sequence, it is safe to have multiple evaluations:

```
sum = (i = 3, i++, i++); // OK: sum = 4, i = 5
```

Each subexpression of the comma expression is evaluated from left to right, and the whole expression evaluates to the rightmost value

The compiler regroups expressions, rearranging associative and commutative operators regardless of parentheses, in order to create an efficiently compiled expression; in no case will the rearrangement affect the value of the expression

3.1.3.4.5.5 Errors And Overflows

Associativity and precedence of CodeGear C++ operators. summarizes the precedence and associativity of the operators. During the evaluation of an expression, the compiler can encounter many problematic situations, such as division by zero or out-of-range floating-point values. Integer overflow is ignored (C uses modulo 2n arithmetic on n-bit registers), but errors detected by math library functions can be handled by standard or user-defined routines.See _matherr and signal.

3.1.3.4.6 Functions

This section contains Function topics.

Topics

Name	Description
Declarations And Definitions (see page 625)	Each program must have a single external function named main or WinMain marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions are **external** by default and are normally accessible from any file in the program. You can restrict visibility of functions by using the **static** storage class specifier (see Linkage (see page 595)). Functions are defined in your source files or made available by linking precompiled libraries. A given function can be declared several times in a program, provided the declarations are compatible. Nondefining function declarations using... more (see page 625)
The main() Function (see page 626)	

Declarations And Prototypes (see page 633)	In the Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows **<type>** func() where **type** is the optional return type defaulting to **int**. In C++, this declaration means <**type**> func(**void**). A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration. This problem was eased by the introduction of function prototypes with... more (see page 633)
Definitions (see page 634)	The general syntax for external function definitions is as follows: **External function definitions**
Extended Types Formatted I/O (see page 634)	The following table shows new format specifiers implemented in C++ Builder for the **printf** and **scanf** family of functions. This implementation allows the input and output of 64-bit integers and provides greater I/O flexibility for other types.
Formal Parameter Declarations (see page 635)	The formal parameter declaration list follows a syntax similar to that of the declarators found in normal identifier declarations. Here are a few examples:
Function Calls And Argument Conversions (see page 635)	A function is called with actual arguments placed in the same sequence as their matching formal parameters. The actual arguments are converted as if by initialization to the declared types of the formal parameters. Here is a summary of the rules governing how the compiler deals with language modifiers and formal parameters in function calls, both with and without prototypes: • The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function. • A function can modify the values of its formal parameters, but this has no effect on... more (see page 635)

3.1.3.4.6.1 Declarations And Definitions

Each program must have a single external function named main or WinMain marking the entry point of the program. Functions are usually declared as prototypes in standard or user-supplied header files, or within program files. Functions are **external** by default and are normally accessible from any file in the program. You can restrict visibility of functions by using the **static** storage class specifier (see Linkage (see page 595)).

Functions are defined in your source files or made available by linking precompiled libraries.

A given function can be declared several times in a program, provided the declarations are compatible. Nondefining function declarations using the function prototype format provide the compiler with detailed parameter information, allowing better control over argument number and type checking, and type conversions.

Note: In C++ you must always use function prototypes. We recommend that you also use them in C.

Excluding C++ function overloading, only one definition of any given function is allowed. The declarations, if any, must also match this definition. (The essential difference between a definition and a declaration is that the definition has a function body.

See Also

Functions

Declarations And Prototypes (see page 633)

Definitions (see page 634)

Forma Parameter Declarations (see page 635)

Function Calls And Argument Conversions (see page 635)

3.1.3.4.6.2 The main() Function

Topics

Name	Description
About The main() Function (see page 627)	Every C and C++ program must have a program-startup function. • Console-based programs call the main function at startup. • Windows GUI programs call the WinMain function at startup. Where you place the startup function is a matter of preference. Some programmers place main at the beginning of the file, others at the end. Regardless of its location, the following points about main always apply. • Arguments to main • Wildcard arguments • Using -p (Pascal Calling Conventions) • The value that main() returns
Arguments to main() (see page 628)	Three parameters (arguments) are passed to main by the CodeGear C++Builder startup routine: **argc**, **argv**, and **env**. • **argc**, an integer, is the number of command-line arguments passed to main, including the name of the executable itself. • **argv** is an array of pointers to strings (char *[])**argv[0]** is the full path name of the program being run.**argv[1**] points to the first string typed on the operating system command line after the program name.**argv[2]** points to the second string typed after the program name.argv[argc-1] points to the last argument passed to main.**argv[argc]** contains... more (see page 628)
Default Runtime Libraries (see page 628)	BCC32.EXE is the command-line compiler. The default libraries used with this compiler are C0W32.OBJ (startup code), CW32.LIB (static single-threaded runtime library), and IMPORT32.LIB (import library for Win32).
Dynamic-link Libraries (see page 629)	The dynamic-link library (DLL) versions of the runtime library are contained in the BIN subdirectory of your installation. These are listed below indicating whether they are multithreaded. ***Directory: BIN***

Multithread Programs (see page 629)	Win32 programs can create more than one thread of execution. If your program creates multiple threads, and these threads also use the C++ runtime library, you must use the CW32MT.LIB or CW32MTI library instead. The multithread libraries provide the following functions which you use to create threads: • _beginthread • _beginthreadex • _beginthreadNT The multithread libraries also provide the following corresponding functions that terminate threads: • _endthread • _endthreadex • _threadid a global variable that contains the current identification number of the thread also known as the **thread ID**). The header file stddef.h contains the declaration of **_threadid**. When you compile or link a... more (see page 629)
Passing File Information To Child Processes (see page 630)	If your program uses the **exec** or **spawn** functions to create a new process, the new process will normally inherit all of the open file handles created by the original process. Some information, however, about these handles will be lost, including the access mode used to open the file. For example, if your program opens a file for read-only access in binary mode, and then spawns a child process, the child process might corrupt the file by writing to it, or by reading from it in text mode. To allow child processes to inherit such information about open files, you... more (see page 630)
Static Runtime Libraries (see page 630)	Listed below is each of the C++Builder static library names and its use. ***Directory of BCB\LIB (LIB files)***
The Value main() Returns (see page 632)	The value returned by **main** is the status code of the program; it must be an **int**. If, however, your program uses the routine **exit** (or **_exit**) to terminate, the value returned by main is the argument passed to the call to exit (or to _exit). For example, if your program contains the call
Using --p (Pascal Calling Conventions) (see page 632)	If you compile your program using Pascal calling conventions, you must remember to explicitly declare main as a C type. Do this with the __cdecl keyword, like this:
Wildcard Arguments (see page 632)	Command-line arguments containing wildcard characters can be expanded to all the matching file names, much the same way DOS expands wildcards when used with commands like COPY. All you have to do to get wildcard expansion is to link your program with the WILDARGS.OBJ object file, which is included with CodeGear C++. **Note:** Wildcard arguments are used only in console-mode applications. Once WILDARGS.OBJ is linked into your program code, you can send wildcard arguments (such as *.*) to your main function. The argument will be expanded (in the argv array) to all files matching the wildcard mask. The maximum... more (see page 632)

3.1.3.4.6.2.1 About The main() Function

Every C and C++ program must have a program-startup function.

- Console-based programs call the main function at startup.
- Windows GUI programs call the WinMain function at startup.

Where you place the startup function is a matter of preference. Some programmers place main at the beginning of the file, others at the end. Regardless of its location, the following points about main always apply.

- Arguments to main
- Wildcard arguments

- Using -p (Pascal Calling Conventions)
- The value that main() returns

See Also

Arguments To Main (see page 628)

Wildcard Arguments (see page 632)

Using -p (Pascal Calling Conventions) (see page 632)

The Value that main() Returns (see page 632)

3.1.3.4.6.2.2 Arguments to main()

Three parameters (arguments) are passed to main by the CodeGear C++Builder startup routine: **argc**, **argv**, and **env**.

- **argc**, an integer, is the number of command-line arguments passed to main, including the name of the executable itself.
- **argv** is an array of pointers to strings (char *[])**argv[0]** is the full path name of the program being run.**argv[1**] points to the first string typed on the operating system command line after the program name.**argv[2]** points to the second string typed after the program name.argv[argc-1] points to the last argument passed to main.**argv[argc]** contains NULL.
- **env** is also an array of pointers to strings. Each element of **env[]** holds a string of the form ENVVAR=value.ENVVAR is the name of an environment variable, such as PATHvalue is the value to which ENVVAR is set, such as C:\APPS;C:\TOOLS.

If you declare any of these parameters, you must declare them exactly in the order given: **argc**, **argv**, **env**. For example, the following are all valid declarations of arguments to **main**:

```
int main()
int main(int argc)      /* legal but very unlikely */
int main(int argc, char * argv[])
it ain(int argc, char * argv[], char * env[])]
```

The declaration int main(int argc) is legal, but it is very unlikely that you would use **argc** in your program without also using the elements of **argv.**

The argument **env** is also available through the global variable _environ..

For all platforms, **argc** and **argv** are also available via the global variables _argc and _argv.

Using main with a Unicode application

The Unicode version of the main function is:

```
int wmain (int argc, wchar_t *argv[])
```

The **argv** (and optional **envp**) parameter(s) support(s) wide-characters.

The following **_tmain** function is a macro that expands to the appropriate version of the main function depending upon the type of application:

```
int _tmain (int argc, _TCHAR *argv[])
```

See Also

The main() Function (see page 627)

3.1.3.4.6.2.3 Default Runtime Libraries

BCC32.EXE is the command-line compiler. The default libraries used with this compiler are C0W32.OBJ (startup code), CW32.LIB (static single-threaded runtime library), and IMPORT32.LIB (import library for Win32).

See Also

The main() Function (see page 627)

3.1.3.4.6.2.4 Dynamic-link Libraries

The dynamic-link library (DLL) versions of the runtime library are contained in the BIN subdirectory of your installation. These are listed below indicating whether they are multithreaded.

Directory: BIN

File Name	
CC3260.DLL	32-bit, single-threaded
CC3260MT.DLL	32-bit, multithreaded

See Also

The main() Function (see page 627)

3.1.3.4.6.2.5 Multithread Programs

Win32 programs can create more than one thread of execution. If your program creates multiple threads, and these threads also use the C++ runtime library, you must use the CW32MT.LIB or CW32MTI library instead.

The multithread libraries provide the following functions which you use to create threads:

- _beginthread
- _beginthreadex
- _beginthreadNT

The multithread libraries also provide the following corresponding functions that terminate threads:

- _endthread
- _endthreadex
- _threadid a global variable that contains the current identification number of the thread also known as the **thread ID**).

The header file stddef.h contains the declaration of _**threadid**.

When you compile or link a program that uses multiple threads, you must use the **-tWM** compiler switch. For example:

```
BCC32 -tWM THREAD.C
```

Note: Take special care when using the signal

function in a multithread program. The SIGINT, SIGTERM, and SIGBREAK signals can be used only by the main thread (thread one) in a non-Win32 application. When one of these signals occurs, the currently executing thread is suspended, and control transfers to the signal handler (if any) set up by thread one. Other signals can be handled by any thread.A signal handler should not use C++ runtime library functions, because a semaphore deadlock might occur. Instead, the handler should simply set a flag or post a semaphore, and return immediately.

See Also

The main() Function (see page 627)

_beginthread (see page 976)

_beginthreadex (see page 980)

_beginthreadNT (see page 978)

_endthread (see page 983)

_endthreadex (see page 984)

_threadid (see page 1005)

signal (see page 1000)

3.1.3.4.6.2.6 Passing File Information To Child Processes

If your program uses the **exec** or **spawn** functions to create a new process, the new process will normally inherit all of the open file handles created by the original process. Some information, however, about these handles will be lost, including the access mode used to open the file. For example, if your program opens a file for read-only access in binary mode, and then spawns a child process, the child process might corrupt the file by writing to it, or by reading from it in text mode.

To allow child processes to inherit such information about open files, you must link your program with the object file FILEINFO.OBJ.

For example:

```
BCC32 TEST.C \BCB\LIB\FILEINFO.OBJ
```

The file information is passed in the environment variable _C_FILE_INFO. This variable contains encoded binary information. Your program should not attempt to read or modify its value. The child program must have been built with the C++ runtime library to inherit this information correctly.

Other programs can ignore _C_FILE_INFO, and will not inherit file information.

See Also

The main() Function (see page 627)

execl (see page 987)

spawnl (see page 989)

3.1.3.4.6.2.7 Static Runtime Libraries

Listed below is each of the C++Builder static library names and its use.

Directory of BCB\LIB (LIB files)

File name	Use
bcbatl.lib	
bcbie.lib	
bcbsmp.lib	
cg32.lib	CodeGuard library (link this in if you want to use CodeGuard)
cghelp.lib	CodeGuard help library
cp32mt.lib	VCL-compliant C RTL multithreaded static library
cp32mti.lib	VCL-compliant import library for multithreaded C RTL cc3260mt.dll
cw32.lib	RTL single-threaded static library
cw32i.lib	Import library for RTL cc3260.dll
cw32mt.lib	RTL multi-threaded static library
cw32mti.lib	Import library for multithreaded RTL cc3260mt.dll
dbx.lib	
dcl31w.lib	
dclact.lib	
dclado.lib	

dclbcbsmp.lib	
dclbde.lib	
dcldb.lib	
dclisp.lib	
dclmid.lib	
dclnet.lib	
dclsoap.lib	
dclstd.lib	
dclwebsnap.lib	
designdgm.lib	
dxextra.lib	DirectX static library
ibevnt.lib	
import32.lib	Import library; includes Winsock 1.x
memmgr.lib	Import library for BORLNDMM.DLL
mswsock.lib	Import library for MSWSOCK.DLL
noeh32.lib	No exception handling support library
obsolete.lib	Old functions
ole2w32.lib	Import library for the 32-bit OLE 2.0 API
oleaut32.lib	Import library for the 32-bit OLE 2.0 API
usebormm.lib	
uuid.lib	GUID static library for miscellaneous Direct 3D, DirectDraw, DirectSound, Shell extensions, DAO, Active Scripting, and so on
vcl.lib	VCL library, contains references to all other VCL libs; use in MAKEFILES with no version #
vcllink.lib	
vclstd.lib	
vcljpg50.lib	Static library for VCL JPEG component
w32inet.lib	Import library for MS Internet DLLs
ws2_32.lib	Import library for the 32-bit WinSock 2.0 API
wininet.lib	Import library for the wininet.dll

Directory of BCB\LIB (OBJ files)

File name	Use
c0d32.obj	DLL startup module
c0d32w.obj	DLL startup module, Wide-char version
c0d32x.obj	DLL startup module, no exception handling
c0pkg32.obj	Package startup module
c0w32.obj	GUI EXE startup module
c0w32w.obj	GUI EXE startup module, Wide-char version

c0x32.obj	32-bit console-mode EXE startup module
c0x32w.obj	32-bit console-mode EXE startup module, Wide-char version
fileinfo.obj	Passes open file-handle information to child processes
gp.obj	Prints register-dump information when an exception occurs
wildargs.obj	Transforms wild-card arguments into an array of arguments to main()/wmain() in console-mode applications

(Lib\PSDK Subdirectory) Import libraries for the Platform SDK

See Also

The main() Function (see page 627)

3.1.3.4.6.2.8 The Value main() Returns

The value returned by **main** is the status code of the program; it must be an **int**. If, however, your program uses the routine **exit** (or _**exit**) to terminate, the value returned by main is the argument passed to the call to exit (or to _exit).

For example, if your program contains the call

```
exit(1)
the status is 1.
```

See Also

About the main() Function (see page 627)

exit (see page 1114)

_exit (see page 1097)

3.1.3.4.6.2.9 Using --p (Pascal Calling Conventions)

If you compile your program using Pascal calling conventions, you must remember to explicitly declare main as a C type. Do this with the __cdecl keyword, like this:

```
int  __cdecl main(int argc, char* argv[], char
* envp[])
```

See Also

The main() Function (see page 627)

3.1.3.4.6.2.10 Wildcard Arguments

Command-line arguments containing wildcard characters can be expanded to all the matching file names, much the same way DOS expands wildcards when used with commands like COPY. All you have to do to get wildcard expansion is to link your program with the WILDARGS.OBJ object file, which is included with CodeGear C++.

Note: Wildcard arguments are used only in console-mode applications.

Once WILDARGS.OBJ is linked into your program code, you can send wildcard arguments (such as *.*) to your main function. The argument will be expanded (in the argv array) to all files matching the wildcard mask. The maximum size of the argv array varies, depending on the amount of memory available in your heap.

If no matching files are found, the argument is passed unchanged. (That is, a string consisting of the wildcard mask is passed to **main**.)

Arguments enclosed in quotes ("...") are not expanded.

See Also

About The Main Function (see page 627)

3.1.3.4.6.3 Declarations And Prototypes

In the Kernighan and Ritchie style of declaration, a function could be implicitly declared by its appearance in a function call, or explicitly declared as follows

<type> func()

where **type** is the optional return type defaulting to **int**. In C++, this declaration means <**type**> func(**void**). A function can be declared to return any type except an array or function type. This approach does not allow the compiler to check that the type or number of arguments used in a function call match the declaration.

This problem was eased by the introduction of function prototypes with the following declaration syntax:

<type> func(parameter-declarator-list);

Note: You can enable a warning within the IDE or with the command-line compiler: "Function called without a prototype."

Declarators specify the type of each function parameter. The compiler uses this information to check function calls for validity. The compiler is also able to coerce arguments to the proper type. Suppose you have the following code fragment:

```
extern long lmax(long v1, long v2); /* prototype */
foo()
{
   int limit = 32;
   char ch = 'A';
   long mval;
   mval = lmax(limit,ch);    /* function call */
}
```

Since it has the function prototype for lmax, this program converts limit and ch to **long**, using the standard rules of assignment, before it places them on the stack for the call to lmax. Without the function prototype, limit and ch would have been placed on the stack as an integer and a character, respectively; in that case, the stack passed to lmax would not match in size or content what lmax was expecting, leading to problems. The classic declaration style does not allow any checking of parameter type or number, so using function prototypes aids greatly in tracking down programming errors.

Function prototypes also aid in documenting code. For example, the function strcpy takes two parameters: a source string and a destination string. The question is, which is which? The function prototype

```
char *strcpy(char *dest, const char *source);
```

makes it clear. If a header file contains function prototypes, then you can print that file to get most of the information you need for writing programs that call those functions. If you include an identifier in a prototype parameter, it is used only for any later error messages involving that parameter; it has no other effect.

A function declarator with parentheses containing the single word **void** indicates a function that takes no arguments at all:

```
func(void);
```

In C++, func() also declares a function taking no arguments

A function prototype normally declares a function as accepting a fixed number of parameters. For functions that accept a variable number of parameters (such as printf), a function prototype can end with an ellipsis (...), like this:

```
f(int *count, long total, ...)
```

With this form of prototype, the fixed parameters are checked at compile time, and the variable parameters are passed with no type checking.

Note: stdarg.h and varargs.h contain macros that you can use in user-defined functions with variable numbers of parameters.

Here are some more examples of function declarators and prototypes:

```
int  f();/* In C, a function returning an int with no information about parameters.
This is the K&R "classic style." */
int f();/* In C++, a function taking no arguments */
int  f(void);/* A function returning an int that takes no parameters. */
int  p(int,long);/* A function returning an int thataccepts two parameters: the first, an int;
the second, a long. */
int  __pascal q(void);/* A pascal function returningan int that takes no parameters at all. */
int  printf(char *format,...;  /*A function returning an int andaccepting a pointer to a char
fixedparameter and any number of additionalparameters of unknown type. */
int  (*fp)(int)/* A pointer to a function returning an intand requiring an int parameter. */
```

3.1.3.4.6.4 Definitions

The general syntax for external function definitions is as follows:

External function definitions

```
file
    external-definition
     file  external-definition
external-definition:
    function-definition
    declaration
    asm-statement
function-definition:
    <declaration-specifiers> declarator <declaration-list>
    compound-statement
```

In general, a function definition consists of the following sections (the grammar allows for more complicated cases):

- 1. Optional storage class specifiers: **extern** or **static**. The default is **extern**.
- 2. A return type, possibly **void**. The default is **int**.
- 3. Optional modifiers: **__pascal**, **__cdecl**, **__export**__saveregs. The defaults depend on the compiler option settings.
- 4. The name of the function.
- 5. A parameter declaration list, possibly empty, enclosed in parentheses. In C, the preferred way of showing an empty list is func(void). The old style of func is legal in C but antiquated and possibly unsafe.
- 6. A function body representing the code to be executed when the function is called.

  Note: You can mix elements from 1 and 2.

3.1.3.4.6.5 Extended Types Formatted I/O

The following table shows new format specifiers implemented in C++ Builder for the **printf** and **scanf** family of functions. This implementation allows the input and output of 64-bit integers and provides greater I/O flexibility for other types.

Format Character	Functionality
%Ld	**__int64**
%I8d	8–bit wide integer (**char**)
%I16d	16–bit wide integer (**short**)
%I32d	32–bit wide integer (**long**)
%I64d	64–bit wide integer (**__int64**)

Note that the above table uses the **%d** format as an example. The **I8, I16, I32, I64** prefixes can be used with the **d, i, o, x, X** formats, as well as the new **L** prefix previously allowed only on **float** to specify **long double** type.

3.1.3.4.6.6 Formal Parameter Declarations

The formal parameter declaration list follows a syntax similar to that of the declarators found in normal identifier declarations. Here are a few examples:

```
int func(void) {                    // no args
int func(T1 t1, T2 t2, T3 t3=1) {  // three simple parameters, one
                                    // with default argument
int func(T1* ptr1, T2& tref) {     // A pointer and a reference arg
int func(register int i) {         // Request register for arg
int func(char *str,...) {          /* One string arg with a variable number of other
                                 args, or with a fixed number of args with varying types */
```

In C++, you can give default arguments as shown. Parameters with default values must be the last arguments in the parameter list. The arguments' types can be scalars, structures, unions, or enumerations; pointers or references to structures and unions; or pointers to functions, classes, or arrays.

The ellipsis (...) indicates that the function will be called with different sets of arguments on different occasions. The ellipsis can follow a sublist of known argument declarations. This form of prototype reduces the amount of checking the compiler can make.

The parameters declared all have automatic scope and duration for the duration of the function. The only legal storage class specifier is **register**.

The **const** and **volatile** modifiers can be used with formal parameter declarators

3.1.3.4.6.7 Function Calls And Argument Conversions

A function is called with actual arguments placed in the same sequence as their matching formal parameters. The actual arguments are converted as if by initialization to the declared types of the formal parameters.

Here is a summary of the rules governing how the compiler deals with language modifiers and formal parameters in function calls, both with and without prototypes:

- The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.
- A function can modify the values of its formal parameters, but this has no effect on the actual arguments in the calling routine, except for reference arguments in C++.

When a function prototype has not been previously declared, the compiler converts integral arguments to a function call according to the integral widening (expansion) rules described in Standard arithmetic conversions. When a function prototype is in scope, the compiler converts the given argument to the type of the declared parameter as if by assignment

When a function prototype includes an ellipsis (...), the compiler converts all given function arguments as in any other prototype (up to the ellipsis). The compiler widens any arguments given beyond the fixed parameters, according to the normal rules for function arguments without prototypes.

If a prototype is present, the number of arguments must match (unless an ellipsis is present in the prototype). The types need to be compatible only to the extent that an assignment can legally convert them. You can always use an explicit cast to convert an argument to a type that is acceptable to a function prototype.

Note: If your function prototype does not match the actual function definition, the compiler will detect this if and only if that definition is in the same compilation unit as the prototype. If you create a library of routines with a corresponding header file of prototypes, consider including that header file when you compile the library, so that any discrepancies between the prototypes and the actual definitions will be caught. C++ provides type-safe linkage, so differences between expected and actual parameters will be caught by the linker.

3.1.3.4.7 Operators Summary

This section contains Operator Summary topics.

Topics

Name	Description
Operators Summary (see page 636)	Operators are tokens that trigger some computation when applied to variables and other objects in an expression. Arithmetic (see page 656) Assignment (see page 591) Bitwise (see page 589) C++ specific (see page 590) Comma (see page 592) Conditional (see page 592) Equality (see page 590) Logical (see page 590) Postfix Expression Operators Primary Expression Operators (see page 645) Preprocessor Reference/Indirect operators (see page 658) Relational (see page 591) sizeof (see page 563) typeid (see page 570) All operators can be overloaded except the following: • . C++ direct component selector • .* C++ dereference • :: C++ scope access/resolution • ?: Conditional Depending on context, the same operator can have more than one meaning. For example, the ampersand (&) can be interpreted as: • a bitwise AND (A & B) • an address operator (&A) • in C++, a reference modifier **Note:** No spaces are... more (see page 636)

3.1.3.4.7.1 Operators Summary

Operators are tokens that trigger some computation when applied to variables and other objects in an expression.

Arithmetic (see page 656)

Assignment (see page 591)

Bitwise (see page 589)

C++ specific (see page 590)

Comma (see page 592)

Conditional (see page 592)

Equality (see page 590)

Logical (see page 590)

Postfix Expression Operators

Primary Expression Operators (see page 645)

Preprocessor

Reference/Indirect operators (see page 658)

Relational (see page 591)

sizeof (see page 563)

typeid (see page 570)

All operators can be overloaded except the following:

- . C++ direct component selector
- .* C++ dereference
- :: C++ scope access/resolution
- ?: Conditional

Depending on context, the same operator can have more than one meaning. For example, the ampersand (&) can be interpreted as:

- a bitwise AND (A & B)
- an address operator (&A)
- in C++, a reference modifier

Note: No spaces are allowed in compound operators. Spaces change the meaning of the operator and will generate an error.

See Also

Precedence Of Operators (see page 622)

3.1.3.4.8 Pointers

This section contains Pointer topics.

Topics

Name	Description
Arrays (see page 638)	The declaration **type** declarator [<constant-expression>] declares an array composed of elements of **type**. An array consists of a contiguous region of storage exactly large enough to hold all of its elements. If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one. Multidimensional arrays are constructed by declaring arrays of array type. The following example shows one way to declare a two-dimensional array. The implementation... more (see page 638)
C++ Reference Declarations (see page 639)	C++ reference types are closely related to pointer types. Reference types create aliases for objects and let you pass arguments to functions by reference. C passes arguments only by value. In C++ you can pass arguments by value or by reference. See Referencing (see page 452) for complete details.
Pointer Arithmetic (see page 639)	Pointer arithmetic is limited to addition, subtraction, and comparison. Arithmetical operations on object pointers of type "pointer to **type**" automatically take into account the size of **type**; that is, the number of bytes needed to store a **type** object. The internal arithmetic performed on pointers depends on the memory model in force and the presence of any overriding pointer modifiers. When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects. Thus, if a pointer is declared to point to **type**, adding an integral value to the pointer advances the pointer... more (see page 639)
Pointer Constants (see page 640)	A pointer or the pointed-at object can be declared with the **const** modifier. Anything declared as a **const** cannot be have its value changed. It is also illegal to create a pointer that might violate the nonassignability of a constant object. Consider the following examples:

Pointer Conversions (see page 640)	Pointer types can be converted to other pointer types using the typecasting mechanism:
Pointer Declarations (see page 641)	A pointer must be declared as pointing to some particular type, even if that type is **void** (which really means a pointer to anything). Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. The compiler lets you reassign pointers like this without typecasting, but the compiler will warn you unless the pointer was originally declared to be of type pointer to void. In C, but not C++, you can assign a **void*** pointer to a non-**void*** pointer. See void for details. **Warning:** You need to initialize pointers... more (see page 641)
Pointers (see page 641)	Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses. The two pointer categories have distinct properties, purposes, and rules for manipulation, although they do share certain characteristics. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory. Although pointers are numbers with most of... more (see page 641)
Pointers To Functions (see page 642)	A pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called. A pointer to a function has a type called "pointer to function returning **type**," where type is the function's return **type**. For example,
Pointers To Objects (see page 642)	A pointer of type "pointer to object of **type**" holds the address of (that is, points to) an object of **type**. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, unions, and classes.

3.1.3.4.8.1 Arrays

The declaration

type declarator [<constant-expression>]

declares an array composed of elements of **type**. An array consists of a contiguous region of storage exactly large enough to hold all of its elements.

If an expression is given in an array declarator, it must evaluate to a positive constant integer. The value is the number of elements in the array. Each of the elements of an array is numbered from 0 through the number of elements minus one.

Multidimensional arrays are constructed by declaring arrays of array type. The following example shows one way to declare a two-dimensional array. The implementation is for three rows and five columns but it can be very easily modified to accept run-time user input.

```
/* DYNAMIC MEMORY ALLOCATION FOR A MULTIDIMENSIONAL OBJECT. */
#include <stdio.h>
#include <stdlib.h>
typedef long double TYPE;
typedef TYPE *OBJECT;
unsigned int rows = 3, columns = 5;
void de_allocate(OBJECT);
int main(void) {
  OBJECT matrix;
  unsigned int i, j;
  /* STEP 1: SET UP THE ROWS. */
  matrix = (OBJECT) calloc( rows, sizeof(TYPE *));
  /* STEP 2: SET UP THE COLUMNS. */
  for (i = 0; i < rows; ++i)
    matrix[i] = (TYPE *) calloc( columns, sizeof(TYPE));
    for (i = 0; i < rows; i++)
      for (j = 0; j < columns; j++)
        matrix[i][j] = i + j;    /* INITIALIZE */
  for (i = 0; i < rows; ++i) {
    printf("\n\n");
```

```
    for (j = 0; j < columns; ++j)
      printf("%5.2Lf", matrix[i][j]);
  de_allocate(matrix);
  return 0;
  }
void de_allocate(OBJECT x) {
  int i;
  for (i = 0; i < rows; i++)    /* STEP 1: DELETE THE COLUMNS */
    free(x[i]);
  free(x);    /* STEP 2: DELETE THE ROWS. */
  }
```

This code produces the following output:

```
0.00 1.00 2.00 3.00 4.00
1.00 2.00 3.00 4.00 5.00
2.00 3.00 4.00 5.00 6.00
```

In certain contexts, the first array declarator of a series might have no expression inside the brackets. Such an array is of indeterminate size. This is legitimate in contexts where the size of the array is not needed to reserve space.

For example, an **extern** declaration of an array object does not need the exact dimension of the array; neither does an array function parameter. As a special extension to ANSI C, CodeGear C++ also allows an array of indeterminate size as the final member of a structure. Such an array does not increase the size of the structure, except that padding can be added to ensure that the array is properly aligned. These structures are normally used in dynamic allocation, and the size of the actual array needed must be explicitly added to the size of the structure in order to properly reserve space.

Except when it is the operand of a **sizeof** or **&** operator, an array type expression is converted to a pointer to the first element of the array.

See Also

Arrays

3.1.3.4.8.2 C++ Reference Declarations

C++ reference types are closely related to pointer types. Reference types create aliases for objects and let you pass arguments to functions by reference. C passes arguments only by value. In C++ you can pass arguments by value or by reference. See Referencing (see page 452) for complete details.

3.1.3.4.8.3 Pointer Arithmetic

Pointer arithmetic is limited to addition, subtraction, and comparison. Arithmetical operations on object pointers of type "pointer to **type**" automatically take into account the size of **type**; that is, the number of bytes needed to store a **type** object.

The internal arithmetic performed on pointers depends on the memory model in force and the presence of any overriding pointer modifiers.

When performing arithmetic with pointers, it is assumed that the pointer points to an array of objects. Thus, if a pointer is declared to point to **type**, adding an integral value to the pointer advances the pointer by that number of objects of **type**. If **type** has size 10 bytes, then adding an integer 5 to a pointer to **type** advances the pointer 50 bytes in memory. The difference has as its value the number of array elements separating the two pointer values. For example, if ptr1 points to the third element of an array, and ptr2 points to the tenth element, then the result of ptr2 - ptr1 would be 7.

The difference between two pointers has meaning only if both pointers point into the same array

When an integral value is added to or subtracted from a "pointer to **type**," the result is also of type "pointer to **type**."

There is no such element as "one past the last element," of course, but a pointer is allowed to assume such a value. If P points to the last array element, P + 1 is legal, but P + 2 is undefined. If P points to one past the last array element, P - 1 is legal, giving a pointer to the last element. However, applying the indirection operator * to a "pointer to one past the last element" leads to

undefined behavior.

Informally, you can think of P + n as advancing the pointer by (n * **sizeof**(**type**)) bytes, as long as the pointer remains within the legal range (first element to one beyond the last element).

Subtracting two pointers to elements of the same array object gives an integral value of type ptrdiff_t defined in stddef.h. This value represents the difference between the subscripts of the two referenced elements, provided it is in the range of ptrdiff_t. In the expression P1 - P2, where P1 and P2 are of type pointer to type (or pointer to qualified type), P1 and P2 must point to existing elements or to one past the last element. If P1 points to the i-th element, and P2 points to the j-th element, P1 - P2 has the value (i - j).

3.1.3.4.8.4 Pointer Constants

A pointer or the pointed-at object can be declared with the **const** modifier. Anything declared as a **const** cannot be have its value changed. It is also illegal to create a pointer that might violate the nonassignability of a constant object. Consider the following examples:

```
int i;                        // i is an int
int * pi;                     // pi is a pointer to int (uninitialized)
int * const cp = &i;          // cp is a constant pointer to int
const int ci = 7;             // ci is a constant int
const int * pci;              // pci is a pointer to constant int
const int * const cpc = &ci; // cpc is a constant pointer to a
                              // constant int
```

The following assignments are legal:

```
i = ci;                      // Assign const-int to int
*cp = ci;                    // Assign const-int to
                             // object-pointed-at-by-a-const-pointer
++pci;                       // Increment a pointer-to-const
pci = cpc;                   // Assign a const-pointer-to-a-const to a
                             // pointer-to-const
```

The following assignments are illegal:

```
ci = 0;                      // NO--cannot assign to a const-int
ci--;                        // NO--cannot change a const-int
*pci = 3;                    // NO--cannot assign to an object
                             // pointed at by pointer-to-const
cp = &ci;                    // NO--cannot assign to a const-pointer,
                             // even if value would be unchanged
cpc++;                       // NO--cannot change const-pointer
pi = pci;                    // NO--if this assignment were allowed,
                             // you would be able to assign to *pci
                             // (a const value) by assigning to *pi.
```

Similar rules apply to the **volatile** modifier. Note that **const** and **volatile** can both appear as modifiers to the same identifier.

3.1.3.4.8.5 Pointer Conversions

Pointer types can be converted to other pointer types using the typecasting mechanism:

```
char *str;
int *ip;
str = (char *)ip;
```

More generally, the cast (**type***) will convert a pointer to type "pointer to **type**."

See C++ specific for a discussion of C++ typecast mechanisms.

3.1.3.4.8.6 Pointer Declarations

A pointer must be declared as pointing to some particular type, even if that type is **void** (which really means a pointer to anything). Once declared, though, a pointer can usually be reassigned so that it points to an object of another type. The compiler lets you reassign pointers like this without typecasting, but the compiler will warn you unless the pointer was originally declared to be of type pointer to void. In C, but not C++, you can assign a **void*** pointer to a non-**void*** pointer. See void for details.

Warning: You need to initialize pointers before using them.

If **type** is any predefined or user-defined type, including **void**, the declaration

```
type *ptr;   /* Uninitialized pointer */
```

declares ptr to be of type "pointer to **type**." All the scoping, duration, and visibility rules apply to the ptr object just declared.

A null pointer value is an address that is guaranteed to be different from any valid pointer in use in a program. Assigning the integer constant 0 to a pointer assigns a null pointer value to it.

The mnemonic NULL (defined in the standard library header files, such as stdio.h) can be used for legibility. All pointers can be successfully tested for equality or inequality to NULL.

The pointer type "pointer to **void**" must not be confused with the null pointer. The declaration

```
void *vptr;
```

declares that vptr is a generic pointer capable of being assigned to by any "pointer to **type**" value, including null, without complaint. Assignments without proper casting between a "pointer to **type1**" and a "pointer to **type2**," where **type1** and **type2** are different types, can invoke a compiler warning or error. If **type1** is a function and **type2** isn't (or vice versa), pointer assignments are illegal. If **type1** is a pointer to **void**, no cast is needed. Under C, if **type2** is a pointer to **void**, no cast is needed.

3.1.3.4.8.7 Pointers

Pointers fall into two main categories: pointers to objects and pointers to functions. Both types of pointers are special objects for holding memory addresses.

The two pointer categories have distinct properties, purposes, and rules for manipulation, although they do share certain characteristics. Generally speaking, pointers to functions are used to access functions and to pass functions as arguments to other functions; performing arithmetic on pointers to functions is not allowed. Pointers to objects, on the other hand, are regularly incremented and decremented as you scan arrays or more complex data structures in memory.

Although pointers are numbers with most of the characteristics of unsigned integers, they have their own rules and restrictions for assignments, conversions, and arithmetic. The examples in the next few sections illustrate these rules and restrictions.

Note: See Referencing for a discussion of referencing and dereferencing.

See Also

Pointers To Objects (see page 642)

Pointers To Functions (see page 642)

Pointer Declarations (see page 641)

Pointer Constants (see page 640)

Pointer Arithmetic (see page 639)

Pointer Conversions (see page 640)

C++ Reference Declarations (see page 639)

3.1.3.4.8.8 Pointers To Functions

A pointer to a function is best thought of as an address, usually in a code segment, where that function's executable code is stored; that is, the address to which control is transferred when that function is called.

A pointer to a function has a type called "pointer to function returning **type**," where type is the function's return **type**. For example,

```
void (*func)();
```

In C++, this is a pointer to a function taking no arguments, and returning **void**. In C, it's a pointer to a function taking an unspecified number of arguments and returning **void**. In this example,

```
void (*func)(int);
```

*func is a pointer to a function taking an **int** argument and returning **void**.

For C++, such a pointer can be used to access static member functions. Pointers to class members must use pointer-to-member operators. See static_cast for details.

3.1.3.4.8.9 Pointers To Objects

A pointer of type "pointer to object of **type**" holds the address of (that is, points to) an object of **type**. Since pointers are objects, you can have a pointer pointing to a pointer (and so on). Other objects commonly pointed at include arrays, structures, unions, and classes.

3.1.3.4.9 Postfix Expression Operators

This section contains Postfix Expression Operator topics.

Topics

Name	Description
. (direct Member Selector) (see page 642)	**Syntax**
-> (indirect Member Selector) (see page 643)	**Syntax**
Array Subscript Operator (see page 643)	Brackets ([ ]) indicate single and multidimensional array subscripts. The expression
Increment/decrement Operators (see page 643)	**Increment operator (++)** Syntax
Function Call Operator (see page 644)	**Syntax**

3.1.3.4.9.1 . (direct Member Selector)

Syntax

```
postfix-expression . identifier
```

postfix-expression must be of type union or structure.

identifier must be the name of a member of that structure or union type.

Remarks

Use the selection operator (.) to access structure and union members.

Suppose that the object s is of struct type S and sptr is a pointer to S. Then, if m is a member identifier of type M declared in S, this expression:

```
s.m
```

are of type M, and represent the member object m in s.

3.1.3.4.9.2 -> (indirect Member Selector)

Syntax

```
postfix-expression -> identifier
```

postfix-expression must be of type pointer to structure or pointer to union.

identifier must be the name of a member of that structure or union type.

The expression designates a member of a structure or union object. The value of the expression is the value of the selected member it will be an lvalue if and only if the postfix expression is an lvalue.

Remarks

You use the selection operator -> to access structure and union members.

Suppose that the object s is of struct type S and sptr is a pointer to S. Then, if m is a member identifier of type M declared in S, this expression:

```
sptr->m
```

is of type M, and represents the member object m in s.

The expression

```
s->sptr
```

is a convenient synonym for (*sptr).m.

3.1.3.4.9.3 Array Subscript Operator

Brackets ([ ]) indicate single and multidimensional array subscripts. The expression

```
<exp1>[exp2]
```

is defined as

```
*((exp1) + (exp2))
```

where either:

- exp1 is a pointer and exp2 is an integer or
- exp1 is an integer and exp2 is a pointer

3.1.3.4.9.4 Increment/decrement Operators

Increment operator (++)

Syntax

```
postfix-expression ++       (postincrement)
++ unary-expression         (preincrement)
```

The expression is called the operand. It must be of scalar type (arithmetic or pointer types) and must be a modifiable lvalue..

Postincrement operator

The value of the whole expression is the value of the postfix expression before the increment is applied.

After the postfix expression is evaluated, the operand is incremented by 1.

Preincrement operator

The operand is incremented by 1 before the expression is evaluated. The value of the whole expression is the incremented value

of the operand.

The increment value is appropriate to the type of the operand.

Pointer types follow the rules for pointer arithmetic.

Decrement operator (--)

Syntax

```
postfix-expression --       (postdecrement)
-- unary-expression         (predecrement)
```

The decrement operator follows the same rules as the increment operator, except that the operand is decremented by 1 after or before the whole expression is evaluated.

3.1.3.4.9.5 Function Call Operator

Syntax

```
postfix-expression(<arg-expression-list>)
```

Remarks

Parentheses ()

- group expressions
- isolate conditional expressions
- indicate function calls and function parameters

The value of the function call expression, if it has a value, is determined by the return statement in the function definition.

This is a call to the function given by the postfix expression.

arg-expression-list is a comma-delimited list of expressions of any type representing the actual (or real) function arguments.

3.1.3.4.10 Primary Expression Operators

This section contains Primary Expression Operator topics.

Topics

Name	Description
Primary Expression Operators (see page 645)	For ANSI C, the primary expressions are literal (also sometimes referred to as constant), identifier, and (expression). The C++ language extends this list of primary expressions to include the keyword **this**, scope resolution operator **::**, name, and the class destructor **~** (tilde). The primary expressions are summarized in the following list. primary-expression: 1. literal **this** (C++ specific) 2. **::** identifier (C++ specific) 3. **::** operator-function-name (C++ specific) 4. **::** qualified-name (C++ specific) 5. (expression) name literal: 1. integer-constantcharacter-constant 2. floating-constant 3. string-literal name: 1. identifieroperator-function-name (C++ specific) 2. conversion-function-name (C++ specific) 3. **~** class-name (C++ specific) 4. qualified-name (C++ specific) qualified-name: (C++ specific) qualified-class-name **::** name For... more (see page 645)

3.1.3.4.10.1 Primary Expression Operators

For ANSI C, the primary expressions are literal (also sometimes referred to as constant), identifier, and (expression). The C++ language extends this list of primary expressions to include the keyword **this**, scope resolution operator **::**, name, and the class destructor **~** (tilde).

The primary expressions are summarized in the following list.

primary-expression:

1. literal **this** (C++ specific)
2. **::** identifier (C++ specific)
3. **::** operator-function-name (C++ specific)
4. **::** qualified-name (C++ specific)
5. (expression) name

literal:

1. integer-constantcharacter-constant
2. floating-constant
3. string-literal

name:

1. identifieroperator-function-name (C++ specific)
2. conversion-function-name (C++ specific)
3. **~** class-name (C++ specific)
4. qualified-name (C++ specific)

qualified-name: (C++ specific)

qualified-class-name **::** name

For a discussion of the primary expression **this**, see this (keyword). The keyword **this** cannot be used outside a class member function body.

The discussion of the scope resolution operator :: begins on page 104. The scope resolution operator allows reference to a type, object, function, or enumerator even though its identifier is hidden.

The parenthesis surrounding an expression do not change the unadorned expression itself.

The primary expression name is restricted to the category of primary expressions that sometimes appear after the member access operators **.** (dot) and **–>** . Therefore, name must be either an lvalue or a function. See also the discussion of member access operators.

An identifier is a primary expression, provided it has been suitably declared. The description and formal definition of identifiers is shown in Lexical Elements: Identifiers.

See the discussion on how to use the destructor operator ~ (tilde).

3.1.3.4.11 Statements

This section contains Statement topics.

Topics

Name	Description
Blocks (see page 647)	A compound statement, or block, is a list (possibly empty) of statements enclosed in matching braces (**{ }**). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth up to the limits of memory.
C++ Specifics (see page 647)	C++ is an object-oriented programming language based on C. Generally speaking, you can compile C programs under C++, but you can't compile a C++ program under C if the program uses any constructs specific to C++. Some situations require special care. For example, the same function func declared twice in C with different argument types causes a duplicated name error. Under C++, however, func will be interpreted as an overloaded function; whether or not this is legal depends on other circumstances. Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside... more (see page 647)
Expression Statements (see page 647)	Any expression followed by a semicolon forms an expression statement:
Iteration Statements (see page 648)	Iteration statements let you loop a set of statements. There are three forms of iteration in C++: **while**, **do while**, and **for** loops.
Jump Statements (see page 648)	A jump statement, when executed, transfers control unconditionally. There are four such statements: **break**, **continue**, **goto**, and **return**

Labeled Statements (see page 648)	A statement can be labeled in two ways: • label-identifier **:** statement * The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and have function scope. In C++ you can label both declaration and non-declaration statements. • **case** constant-expression **:** statement * **default :** statement Case and default labeled statements are used only in conjunction with switch statements.
Selection Statements (see page 648)	Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the **if...else** and the **switch**.
Statements (see page 648)	Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. CodeGear C++ statements shows the syntax for statements. **CodeGear C++ statements**

3.1.3.4.11.1 Blocks

A compound statement, or block, is a list (possibly empty) of statements enclosed in matching braces (**{ }**). Syntactically, a block can be considered to be a single statement, but it also plays a role in the scoping of identifiers. An identifier declared within a block has a scope starting at the point of declaration and ending at the closing brace. Blocks can be nested to any depth up to the limits of memory.

3.1.3.4.11.2 C++ Specifics

C++ is an object-oriented programming language based on C. Generally speaking, you can compile C programs under C++, but you can't compile a C++ program under C if the program uses any constructs specific to C++. Some situations require special care. For example, the same function func declared twice in C with different argument types causes a duplicated name error. Under C++, however, func will be interpreted as an overloaded function; whether or not this is legal depends on other circumstances.

Although C++ introduces new keywords and operators to handle classes, some of the capabilities of C++ have applications outside of any class context. This topic discusses the aspects of C++ that can be used independently of classes, then describes the specifics of classes and class mechanisms.

See C++ Exception Handling and C-Based Structured Exceptions for details on compiling C and C++ programs with exception handling.

See Also

Referencing (see page 452)

C++ Classes (see page 387)

Introduction To Constructors And Destructors (see page 415)

Polymorphic Classes (see page 447)

C++ Scope (see page 430)

3.1.3.4.11.3 Expression Statements

Any expression followed by a semicolon forms an expression statement:

```
<expression>;
```

The compiler executes an expression statement by evaluating the expression. All side effects from this evaluation are completed before the next statement is executed. Most expression statements are assignment statements or function calls

The null statement is a special case, consisting of a single semicolon (**;**). The null statement does nothing, and is therefore useful in situations where C++ syntax expects a statement but your program does not need one.

3.1.3.4.11.4 Iteration Statements

Iteration statements let you loop a set of statements. There are three forms of iteration in C++: **while**, **do while**, and **for** loops.

3.1.3.4.11.5 Jump Statements

A jump statement, when executed, transfers control unconditionally. There are four such statements: **break**, **continue**, **goto**, and **return**

3.1.3.4.11.6 Labeled Statements

A statement can be labeled in two ways:

- label-identifier **:** statement

* The label identifier serves as a target for the unconditional **goto** statement. Label identifiers have their own name space and have function scope. In C++ you can label both declaration and non-declaration statements.

- **case** constant-expression **:** statement

* **default :** statement

Case and default labeled statements are used only in conjunction with switch statements.

3.1.3.4.11.7 Selection Statements

Selection or flow-control statements select from alternative courses of action by testing certain values. There are two types of selection statements: the **if...else** and the **switch**.

3.1.3.4.11.8 Statements

Statements specify the flow of control as a program executes. In the absence of specific jump and selection statements, statements are executed sequentially in the order of appearance in the source code. CodeGear C++ statements shows the syntax for statements.

CodeGear C++ statements

statement	labeled-statement compound-statement expression-statement selection-statement iteration-statement jump-statement asm-statement declaration (C++ specific)
labeled-statement:	identifier : statement **case** constant-expression : statement **default** : statement

compound-statement:	{ <declaration-list> <statement-list> }
declaration-list:	declaration declaration-list declaration
statement-list:	statement statement-list statement
expression-statement:	<expression> ;
asm-statement:	**asm** tokens newline **asm** tokens; **asm** { tokens; <tokens;>= <tokens;>}
selection-statement:	**if** (expression) statement **if** (expression) statement else statement **switch** (expression) statement
iteration-statement:	**while** (expression) statement **do** statement while (expression) ; **for** (for-init-statement <expression> ; <expression>) statement
for-init-statement:	expression-statement declaration (C++ specific)
jump-statement:	**goto** identifier ; **continue** ; **break** ; **return** <expression> ;

3.1.3.4.12 Structures

This section contains Structure topics.

Topics

Name	Description
Structures (see page 650)	A structure is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure member can be a bit field type not allowed elsewhere. The structure type lets you handle complex data structures almost as easily as single variables. Structure initialization is discussed in Arrays, structures, and unions in the help topic Initialization (see page 611). In C++, a structure type is treated as a class type with certain differences: default access is public, and... more (see page 650)
Untagged Structures And Typedefs (see page 650)	If you omit the structure tag, you can get an untagged structure. You can use untagged structures to declare the identifiers in the comma-delimited struct-id-list to be of the given structure type (or derived from it), but you cannot declare additional objects of this type elsewhere
Structure Member Declarations (see page 651)	The member-decl-list within the braces declares the types and names of the structure members using the declarator syntax shown in CodeGear C++ declaration syntax. A structure member can be of any type, with two exceptions The member type cannot be the same as the **struct** type being currently declared:
Structures And Functions (see page 651)	A function can return a structure type or a pointer to a structure type:

Structure Member Access (see page 651)	Structure and union members are accessed using the following two selection operators: • . (period) • -> (right arrow) Suppose that the object s is of struct type S, and sptr is a pointer to S. Then if m is a member identifier of type M declared in S, the expressions s.m and sptr->m are of type M, and both represent the member object m in S. The expression sptr->m is a convenient synonym for (*sptr).m. The operator . is called the direct member selector and the operator -> is called the indirect (or pointer) member selector. For example:
Structure Name Spaces (see page 652)	Structure tag names share the same name space with union tags and enumeration tags (but **enums** within a structure are in a different name space in C++). This means that such tags must be uniquely named within the same scope. However, tag names need not differ from identifiers in the other three name spaces: the label name space, the member name space(s), and the single name space (which consists of variables, functions, **typedef** names, and enumerators). Member names within a given structure or union must be unique, but they can share the names of members in other structures or unions.... more (see page 652)
Incomplete Declarations (see page 653)	Incomplete declarations are also known as forward declarations. A pointer to a structure type A can legally appear in the declaration of another structure B before A has been declared:
Bit Fields (see page 653)	Bit fields are specified numbers of bits that may or may not have an associated identifier. Bit fields offer a way of subdividing structures (structs, unions, classes) into named parts of user-defined sizes. **Declaring bit fields** You specify the bit-field width and optional identifier as follows:

3.1.3.4.12.1 Structures

A structure is a derived type usually representing a user-defined collection of named members (or components). The members can be of any type, either fundamental or derived (with some restrictions to be noted later), in any sequence. In addition, a structure member can be a bit field type not allowed elsewhere. The structure type lets you handle complex data structures almost as easily as single variables. Structure initialization is discussed in Arrays, structures, and unions in the help topic Initialization (see page 611).

In C++, a structure type is treated as a class type with certain differences: default access is public, and the default for the base class is also public. This allows more sophisticated control over access to structure members by using the C++ access specifiers: **public** (the default), **private**, and **protected**. Apart from these optional access mechanisms, and from exceptions as noted, the following discussion on structure syntax and usage applies equally to C and C++ structures.

Structures are declared using the keyword **struct**. For example

```
struct mystruct { ... }; // mystruct is the structure tag
    .
    .
    .
struct mystruct s, *ps, arrs[10];
/* s is type struct mystruct; ps is type pointer to struct mystruct;
   arrs is array of struct mystruct. */
```

3.1.3.4.12.2 Untagged Structures And Typedefs

If you omit the structure tag, you can get an untagged structure. You can use untagged structures to declare the identifiers in the comma-delimited struct-id-list to be of the given structure type (or derived from it), but you cannot declare additional objects of

this type elsewhere

```
struct { ... } s, *ps, arrs[10]; // untagged structure
```

It is possible to create a **typedef** while declaring a structure, with or without a tag:

```
typedef struct mystruct { ... } MYSTRUCT;
MYSTRUCT s, *ps, arrs[10];       // same as struct mystruct s, etc.
typedef struct { ... } YRSTRUCT; // no tag
YRSTRUCT y, *yp, arry[20];
```

Usually, you don't need both a tag and a **typedef**: either can be used in structure declarations.

Untagged structure and union members are ignored during initialization.

3.1.3.4.12.3 Structure Member Declarations

The member-decl-list within the braces declares the types and names of the structure members using the declarator syntax shown in CodeGear C++ declaration syntax.

A structure member can be of any type, with two exceptions

The member type cannot be the same as the **struct** type being currently declared:

```
struct mystruct { mystruct s } s1, s2; // illegal
```

However, a member can be a pointer to the structure being declared, as in the following example:

```
struct mystruct { mystruct *ps } s1, s2; // OK
```

Also, a structure can contain previously defined structure types when declaring an instance of a declared structure.

Except in C++, a member cannot have the type "function returning...," but the type "pointer to function returning..." is allowed. In C++, a **struct** can have member functions.

Note: You can omit the struct

keyword in C++.

3.1.3.4.12.4 Structures And Functions

A function can return a structure type or a pointer to a structure type:

```
mystruct func1(void);  // func1() returns a structure
mystruct *func2(void); // func2() returns pointer to structure
```

A structure can be passed as an argument to a function in the following ways:

```
void func1(mystruct s);        // directly
void func2(mystruct *sptr);    // via a pointer
void func3(mystruct &sref);    // as a reference (C++ only)
```

3.1.3.4.12.5 Structure Member Access

Structure and union members are accessed using the following two selection operators:

- . (period)
- -> (right arrow)

Suppose that the object s is of struct type S, and sptr is a pointer to S. Then if m is a member identifier of type M declared in S, the expressions s.m and sptr->m are of type M, and both represent the member object m in S. The expression sptr->m is a convenient synonym for (*sptr).m.

The operator . is called the direct member selector and the operator -> is called the indirect (or pointer) member selector. For example:

```
struct mystruct
{
   int i;
   char str[21];
   double d;
} s, *sptr = &s;
    .
    .
    .
s.i = 3;            // assign to the i member of mystruct s
sptr -> d = 1.23;   // assign to the d member of mystruct s
```

The expression s.m is an lvalue, provided that s is an lvalue and m is not an array type. The expression sptr->m is an lvalue unless m is an array type.

If structure B contains a field whose type is structure A, the members of A can be accessed by two applications of the member selectors

```
struct A {
   int j;
   double x;
};
struct B {
   int i;
   struct A a;
   double d;
} s, *sptr;
    .
    .
    .
s.i = 3;            // assign to the i member of B
s.a.j = 2;          // assign to the j member of A
sptr->d = 1.23;     // assign to the d member of B
sptr->a.x = 3.14  // assign to x member of A
```

Each structure declaration introduces a unique structure type, so that in

```
struct A {
   int i,j;
   double d;
} a, a1;
struct B {
   int i,j;
   double d;
} b;
```

the objects a and a1 are both of type struct A, but the objects a and b are of different structure types. Structures can be assigned only if the source and destination have the same type:

```
a = a1;   // OK: same type, so member by member assignment
a = b;    // ILLEGAL: different types
a.i = b.i; a.j = b.j; a.d = b.d /* but you can assign member-by-member */
```

3.1.3.4.12.6 Structure Name Spaces

Structure tag names share the same name space with union tags and enumeration tags (but **enums** within a structure are in a different name space in C++). This means that such tags must be uniquely named within the same scope. However, tag names need not differ from identifiers in the other three name spaces: the label name space, the member name space(s), and the single name space (which consists of variables, functions, **typedef** names, and enumerators).

Member names within a given structure or union must be unique, but they can share the names of members in other structures or unions. For example

```
goto s;
    .
    .
```

```
    .
s:          // Label
struct s {  // OK: tag and label name spaces different
   int s;   // OK: label, tag and member name spaces different
   float s; // ILLEGAL: member name duplicated
} s;        // OK: var name space different. In C++, this can only
            // be done if s does not have a constructor.
union s {   // ILLEGAL: tag space duplicate
   int s;   // OK: new member space
   float f;
} f;        // OK: var name space
struct t {
   int s;   // OK: different member space
    .
    .
    .
} s;        // ILLEGAL: var name duplicate
```

3.1.3.4.12.7 Incomplete Declarations

Incomplete declarations are also known as forward declarations.

A pointer to a structure type A can legally appear in the declaration of another structure B before A has been declared:

```
struct A;                  // incomplete
struct B { struct A *pa };
struct A { struct B *pb };
```

The first appearance of A is called incomplete because there is no definition for it at that point. An incomplete declaration is allowed here, because the definition of B doesn't need the size of A.

3.1.3.4.12.8 Bit Fields

Bit fields are specified numbers of bits that may or may not have an associated identifier. Bit fields offer a way of subdividing structures (structs, unions, classes) into named parts of user-defined sizes.

Declaring bit fields

You specify the bit-field width and optional identifier as follows:

```
type-specifier <bitfield-id> : width;
```

In C++, type-specifier is **bool**, **char**, **unsigned char**, **short**, **unsigned short**, **long**, **unsigned long**, **int**, **unsigned int,** **__int64** or **unsigned** **__int64**. In strict ANSI C, type-specifier is **int** or **unsigned int**.

The expression width must be present and must evaluate to a constant integer. In C++, the width of a bit field may be declared of any size. In strict ANSI C, the width of a bit field may be declared only up to the size of the declared type. A zero length bit field skips to the next allocation unit.

If the bit field identifier is omitted, the number of bits specified in width is allocated, but the field is not accessible. This lets you match bit patterns in, say, hardware registers where some bits are unused.

Bit fields can be declared only in structures, unions, and classes. They are accessed with the same member selectors (. and ->) used for non-bit-field members.

Limitations of using bit fields

When using bit fields, be aware of the following issues:

- The code will be non-portable since the organization of bits-within-bytes and bytes-within-words is machine dependent.
- You cannot take the address of a bit field; so the expression &mystruct.x is illegal if x is a bit field identifier, because there is no guarantee that mystruct.x lies at a byte address.
- Bit fields are used to pack more variables into a smaller data space, but causes the compiler to generate additional code to

manipulate these variables. This costs in terms of code size and execution time.

Because of these disadvantages, using bit fields is generally discouraged, except for certain low-level programming. A recommended alternative to having one-bit variables, or flags, is to use defines. For example:

#define Nothing 0x00

#define bitOne 0x01

#define bitTwo 0x02

#define bitThree 0x04

#define bitFour 0x08

#define bitFive 0x10

#define bitSix 0x20

#define bitSeven 0x40

#define bitEight 0x80

can be used to write code like:

```
if (flags & bitOne) {...}    // is bit One turned on
flags |= bitTwo;               // turn bit Two on
flags &= ~bitThree;          // turn bit Three off
```

Similar schemes can be made for bit fields of any size.

Padding of bit fields

In C++, if the width size is larger than the type of the bit field, the compiler will insert padding equal to the requested width size minus the size of the type of the bit field. So, declaration:

```
struct mystruct
{
  int i : 40;
  int j : 8;
};
```

will create a 32 bit storage for 'i', an 8 bit padding, and 8 bit storage for 'j'. To optimize access, the compiler will consider 'i' to be a regular int variable, not a bit field.

Layout and alignment

Bit fields are broken up into groups of consecutive bit fields of the same type, without regard to signedness. Each group of bit fields is aligned to the current alignment of the type of the members of the group. This alignment is determined by the type AND by the setting of the overall alignment (set by the byte alignment option –aN). Within each group, the compiler will pack the bit fields inside of areas as large as the type size of the bit fields. However, no bit field is allowed to straddle the boundary between 2 of those areas. The size of the total structure will be aligned, as determined by the current alignment.

Example of bit field layout, padding, and alignment

In the following C++ declaration, my_struct contains 6 bit fields of 3 different types, **int**, **long**, and **char**:

```
struct my_struct
{
   int one : 8;
   unsigned int two : 16;
   unsigned long three : 8;
   long four : 16;
   long five : 16;
   char six : 4;
};
```

Bit fields 'one' and 'two' will be packed into one 32-bit area.

Next, the compiler inserts padding, if necessary, based on the current alignment, and the type of three, because the type

changes between the declarations of variables two and three. For example, if the current alignment is byte alignment (-a1), no padding is needed; whereas, if the alignment is 4 bytes (-a4), then 8-bit padding is inserted.

Next, variables three, four, and five are all of type **long**. Variables three and four are packed into one 32 bit area, but five can not be packed into that same area, since that would create an area of 40 bits, which is more than the 32 bit allowed for the **long** type. To start a new area for five, the compiler would insert no padding if the current alignment is byte alignment, or would insert 8 bits of padding if the current alignment is dword (4 byte) alignment.

With variable six, the type changes again. Since **char** is always byte aligned, no padding is needed. To force alignment for the whole struct, the compiler will finish up the last area with 4 bits of padding if byte alignment is used, or 12 bits of padding if dword alignment is used.

The total size of my_struct is 9 bytes with byte alignment, or 12 bytes with dword alignment.

To get the best results when using bit fields you should:

- Sort bit fields by type
- Make sure they are packed inside the areas by ordering them such that no bit field will straddle an area boundary
- Make sure the struct is filled as much as possible.

Another recommendation is to force byte alignment for this struct, by emitting "#pragma option -a1". If you want to know how big your struct is, follow it by "#pragma sizeof(mystruct)", which gives you the size.

Using one bit signed fields

For a signed type of one bit, the possible values are 0 or –1. For an unsigned type of one bit, the possible values are 0 or 1. Note that if you assign “1” to a signed bit field, the value will be evaluated as –1 (negative one).

When storing the values **true** and **false** into a one bit sized bit field of a signed type, you can not test for equality to **true** because signed one bit sized bit fields can only hold the values '0' and '-1', which are not compatible with **true** and **false**. You can, however, test for non-zero.

For unsigned varieties of all types, and of course for the **bool** type, testing for equality to **true** will work as expected.

Thus:

```
struct mystruct
{
   int flag : 1;
} M;
int testing()
{
   M.flag = true;
   if(M.flag == true)
     printf("success");}
```

will NOT work, but:

```
struct mystruct
{
   int flag : 1;
} M;
int testing()
{
   M.flag = true;
   if(M.flag)
     printf("success");
}
```

works just fine.

Notes on compatibility

Between versions of the compiler, changes can be made to default alignment, or for purposes of compatibility with other compilers. Consequently, this could change the alignment of bit fields. Therefore, there is no guarantee that the alignment of bit

fields will be consistent between versions of the compiler. To check the backward compatibility of bit fields in your code you can add an assert statement that checks for the structure size you are expecting.

According to the C and C++ language specifications, the alignment and storage of bit fields is implementation defined. Therefore, compilers can align and store bit fields differently. If you want complete control over the layout of bit fields, it is advisable to write your own bit field accessing routines and create your own bit fields.

3.1.3.4.13 Unary Operators

This section contains C++ Unary Operator topics.

Topics

Name	Description
Unary Operators (see page 656)	**Syntax**
Arithmetic Operators (see page 656)	**Syntax**
Plus And Minus Operators (see page 657)	**Unary** In these unary + - expressions
Reference/Deference Operators (see page 658)	**Syntax**

3.1.3.4.13.1 Unary Operators

Syntax

```
<unary-operator> <unary expression>
```

OR

```
<unary-operator> <type><unary expression>
```

Remarks

Unary operators group right-to-left.

The C++ language provides the following unary operators:

- ! Logical negation
- * Indirection
- ++ Increment
- ~ Bitwise complement
- -- Decrement
- - Unary minus
- + Unary plus

3.1.3.4.13.2 Arithmetic Operators

Syntax

```
+ cast-expression
- cast-expression
add-expression + multiplicative-expression
add-expression - multiplicative-expression
multiplicative-expr * cast-expr
multiplicative-expr / cast-expr
multiplicative-expr % cast-expr
postfix-expression ++       (postincrement)
++ unary-expression         (preincrement)
postfix-expression --       (postdecrement)
-- unary-expression         (predecrement)
```

Remarks

Use the arithmetic operators to perform mathematical computations.

The unary expressions of + and - assign a positive or negative value to the cast-expression.

- (addition), - (subtraction), * (multiplication), and / (division) perform their basic algebraic arithmetic on all data types, integer and floating point.

% (modulus operator) returns the remainder of integer division and cannot be used with floating points.

++ (increment) adds one to the value of the expression. Postincrement adds one to the value of the expression after it evaluates; while preincrement adds one before it evaluates.

-- (decrement) subtracts one from the value of the expression. Postdecrement subtracts one from the value of the expression after it evaluates; while predecrement subtracts one before it evaluates.

3.1.3.4.13.3 Plus And Minus Operators

Unary

In these unary + - expressions

```
+ cast-expression
- cast-expression
```

the cast-expression operand must be of arithmetic type.

Results

- cast-expression Value of the operand after any required integral promotions.
- cast-expression Negative of the value of the operand after any required integral promotions.

Binary

Syntax

```
add-expression + multiplicative-expression
add-expression - multiplicative-expression
```

Legal operand types for op1 + op2:

- 1. Both op1 and op2 are of arithmetic type.
- 2. op1 is of integral type, and op2 is of pointer to object type.
- 3. op2 is of integral type, and op1 is of pointer to object type.

In case 1, the operands are subjected to the standard arithmetical conversions, and the result is the arithmetical sum of the operands.

In cases 2 and 3, the rules of pointer arithmetic apply.

Legal operand types for op1 - op2:

- 1. Both op1 and op2 are of arithmetic type.
- 2. Both op1 and op2 are pointers to compatible object types.
- 3. op1 is of pointer to object type, and op2 is integral type.

In case 1, the operands are subjected to the standard arithmetic conversions, and the result is the arithmetic difference of the operands.

In cases 2 and 3, the rules of pointer arithmetic apply.

Note: The unqualified type <type> is considered to be compatible with the qualified types const type, volatile type,and const volatile type.

3.1.3.4.13.4 Reference/Deference Operators

Syntax

```
& cast-expression
* cast-expression
```

Remarks

The **&** and ***** operators work together to reference and dereference pointers that are passed to functions.

Referencing operator (&)

Use the reference operator to pass the address of a pointer to a function outside of main().

The cast-expression operand must be one of the following:

- a function designator
- an lvalue designating an object that is not a bit field and is not declared with a register storage class specifier

If the operand is of type <type>, the result is of type pointer to <type>.

Some non-lvalue identifiers, such as function names and array names, are automatically converted into “pointer-to-X” types when they appear in certain contexts. The **&** operator can be used with such objects, but its use is redundant and therefore discouraged.

Consider the following example:

```
T t1 = 1, t2 = 2;
T *ptr = &t1;      // Initialized pointer
*ptr = t2;           // Same effect as t1 = t2
```

T *ptr = &t1 is treated as

```
T *ptr;
ptr = &t1;
```

So it is ptr, or *ptr, that gets assigned. Once ptr has been initialized with the address &t1, it can be safely dereferenced to give the lvalue *ptr.

Indirection operator (*)

Use the asterisk (*****) in a variable expression to create pointers. And use the indirect operator in external functions to get a pointer's value that was passed by reference.

If the operand is of type pointer to function, the result is a function designator.

If the operand is a pointer to an object, the result is an lvalue designating that object.

The result of indirection is undefined if either of the following occur:

- 1. The cast-expression is a null pointer.
- 2. The cast-expression is the address of an automatic variable and execution of its block has terminated.

Note: & can also be the bitwise AND operator.

Note: * can also be the multiplication operator.

3.1.3.4.14 Unions

This section contains Union topics.

Topics

Name	Description
Unions (see page 659)	Union types are derived types sharing many of the syntactic and functional features of structure types. The key difference is that a union allows only one of its members to be "active" at any one time. The size of a union is the size of its largest member. The value of only one of its members can be stored at any time. In the following simple case,
Anonymous Unions (see page 659)	A union that does not have a tag and is not used to declare a named object (or other type) is called an anonymous union. It has the following form:
Union Declarations (see page 660)	The general declaration syntax for unions is similar to that for structures. The differences are: • Unions can contain bit fields, but only one can be active. They all start at the beginning of the union. (Note that, because bit fields are machine dependent, they can pose problems when writing portable code.) • Unlike C++ structures, C++ union types cannot use the class access specifiers: **public**, **private**, and **protected**. All fields of a union are public. • Unions can be initialized only through their first declared member:

3.1.3.4.14.1 Unions

Union types are derived types sharing many of the syntactic and functional features of structure types. The key difference is that a union allows only one of its members to be "active" at any one time. The size of a union is the size of its largest member. The value of only one of its members can be stored at any time. In the following simple case,

```
union myunion {     /* union tag = myunion */
   int i;
   double d;
   char ch;
} mu, *muptr=&mu;
```

the identifier mu, of type **union** myunion, can be used to hold an **int**, an 8-byte **double**, or a single-byte **char**, but only one of these at the same time

Note: Unions correspond to the variant record types of Delphi.

sizeof(**union** myunion) and **sizeof**(mu) both return 8, but 4 bytes are unused (padded) when mu holds an **int** object, and 7 bytes are unused when mu holds a **char**. You access union members with the structure member selectors (. and ->), but care is needed:

```
mu.d = 4.016;
printf("mu.d = %f\n",mu.d); //OK: displays mu.d = 4.016printf("mu.i = %d\n",mu.i); //peculiar resultmu.ch = 'A';
printf("mu.ch = %c\n",mu.ch); //OK: displays mu.ch = Aprintf("mu.d = %f\n",mu.d); //peculiar resultmuptr->i = 3;
printf("mu.i = %d\n",mu.i); //OK: displays mu.i = 3
```

The second printf is legal, since mu.i is an integer type. However, the bit pattern in mu.i corresponds to parts of the **double** previously assigned, and will not usually provide a useful integer interpretation.

When properly converted, a pointer to a union points to each of its members, and vice versa.

3.1.3.4.14.2 Anonymous Unions

A union that does not have a tag and is not used to declare a named object (or other type) is called an anonymous union. It has the following form:

```
union { member-list };
```

Its members can be accessed directly in the scope where this union is declared, without using the x.y or p->y syntax.

Anonymous unions can be global, nested, or unnested. ANSI-C never allows anonymous unions. ANSI-C++ allows all three types of anonymous unions.

C++ Anonymous unions

An anonymous union cannot have member functions or private or protected members. At file level an anonymous union must be declared static. Local anonymous unions must be either automatic or static; in other words, an anonymous union cannot have external linkage. Unnested anonymous unions are only allowed in C++.

Nested anonymous unions

The outer structure, class, or union of a nested anonymous union must have a tag. CodeGear C and C++ allow nested anonymous unions by default. In C only, a nested anonymous union can, optionally, have a tag.

Example

```
struct outer {
        int x;
};
int main(void)
{
        struct outer o;
}
```

3.1.3.4.14.3 Union Declarations

The general declaration syntax for unions is similar to that for structures. The differences are:

- Unions can contain bit fields, but only one can be active. They all start at the beginning of the union. (Note that, because bit fields are machine dependent, they can pose problems when writing portable code.)
- Unlike C++ structures, C++ union types cannot use the class access specifiers: **public**, **private**, and **protected**. All fields of a union are public.
- Unions can be initialized only through their first declared member:

```
union local87 {
  int i;
  double d;
  } a = { 20 };
```

A union can't participate in a class hierarchy. It can't be derived from any class, nor can it be a base class. A union can have a constructor.

3.1.3.5 Lexical Elements

This section contains Lexical Element topics.

Topics

Name	Description
Lexical Elements (see page 661)	These topics provide a formal definition of C++ lexical elements. They describe the different categories of word-like units (tokens) recognized by a language. The tokens in a C++ source file are derived from a series of operations performed on your programs by the compiler and its built-in preprocessor. The preprocessor first scans the program text for special preprocessor directives (see Preprocessor directives for details). For example, the directive **#include** <inc_file> adds (or includes) the contents of the file inc_file to the program before the compilation phase. The preprocessor also expands any macros found in the program and include files. A... more (see page 661)

Tokens Overview (see page 661)	This section contains Token topics.
Whitespace Overview (see page 684)	This section contains Whitespace Overview topics.

3.1.3.5.1 Lexical Elements

These topics provide a formal definition of C++ lexical elements. They describe the different categories of word-like units (tokens) recognized by a language.

The tokens in a C++ source file are derived from a series of operations performed on your programs by the compiler and its built-in preprocessor.

The preprocessor first scans the program text for special preprocessor directives (see Preprocessor directives for details). For example, the directive **#include** <inc_file> adds (or includes) the contents of the file inc_file to the program before the compilation phase. The preprocessor also expands any macros found in the program and include files.

A C++ program starts as a sequence of ASCII characters representing the source code, created using a suitable text editor (such as the IDE's editor). The basic program unit in C++ is a source file (usually designated by a ".c", or ".cpp" in its name), and all of the header files and other source files included with the **#include** preprocessor directive. Source files are usually designated by a ".c" or ".cpp" in the name, while header files are usually designated with a ".h" or ".hpp".

In the tokenizing phase of compilation, the source code file is parsed (that is, broken down) into tokens and whitespace.

See Also

Whitespace (see page 686)

Tokens (see page 678)

3.1.3.5.2 Tokens Overview

This section contains Token topics.

Topics

Name	Description
Constants Overview (see page 662)	This section contains Constant topics.
Tokens (see page 678)	Tokens are word-like units recognized by a language. The compiler recognizes six classes of tokens. Here is the formal definition of a token: • keyword • identifier • constant • string-literal • operator • punctuator (also known as separators) As the source code is scanned, tokens are extracted in such a way that the longest possible token from the character sequence is selected. For example, external would be parsed as a single identifier, rather than as the keyword **extern** followed by the identifier al. See Token Pasting with ## for a description of token pasting.
Identifiers Overview (see page 679)	This section contains Identifier topics.
Keywords Overview (see page 680)	This section contains Keyword topics.
Punctuators Overview (see page 681)	This section contains Punctuator topics.

3.1.3.5.2.1 Constants Overview

This section contains Constant topics.

Topics

Name	Description
Character Constants Overview (see page 662)	This section contains Character Constant topics.
Constants (see page 667)	Constants are tokens representing fixed numeric or character values. The compiler supports four classes of constants: integer, floating point, character (including strings), and enumeration. Internal representation of numerical types shows how these types are represented internally. The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code. The formal definition of a constant is shown in the following table. ***Constants: Formal Definitions***
Integer Constant Without L Or U (see page 669)	***Decimal constants***
Constant Expressions (see page 670)	This section contains Constant Expression topics.
Constants and Internal Representation (see page 670)	This section contains Constants and Internal Representation topics.
Floating Point Constants (see page 672)	A floating-point constant consists of: • Decimal integer • Decimal point • Decimal fraction • e or E and a signed integer exponent (optional) • Type suffix: f or F or l or L (optional) You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter e (or E) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations. Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed. Here are some examples:
Enumeration Constants (see page 673)	This section contains Enumeration Constant topics.
Integer Constants (see page 674)	Integer constants can be decimal (base 10), octal (base 8) or hexadecimal (base 16). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value, as shown in Integer constants without L or U.. Note that the rules vary between decimal and nondecimal constants. **Decimal** Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit are truncated. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,
Internal Representation of Numerical Types (see page 675)	This section contains Internal Representation of Numerical Type topics.
String Constants (see page 677)	This section contains String Constant topics.

3.1.3.5.2.1.1 Character Constants Overview

This section contains Character Constant topics.

Topics

Name	Description
Character Constants (see page 663)	A character constant is one or more characters enclosed in single quotes, such as 'A', '+', or '\n'. In C, single-character constants have data type **int**. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**. To learn more about character constants, see the following topics. • Three char types (see page 666) • Escape sequences (see page 664) • Wide-character and multi-character constants (see page 666) **Note:** To compare sizes of character types, compile this as a C program and then as a C++ program.
Escape Sequences (see page 664)	The backslash character (\) is used to introduce an escape sequence, which allows the visual representation of certain nongraphic characters. For example, the constant **\n** is used to the single newline character. A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that value; for example, '\03' for Ctrl-C or '\x3F' for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type **char** (0 to 0xff). Larger... more (see page 664)
The Three Char Types (see page 666)	One-character constants, such as 'A', '\t' and '007', are represented as **int** values. In this case, the low-order byte is sign extended into the high bit; that is, if the value is greater than 127 (base 10), the upper bit is set to -1 (=0xFF). This can be disabled by declaring that the default **char** type is **unsigned.** The three character types, **char**, **signed char**, and **unsigned char,** require an 8-bit (one byte) storage. By default, the compiler treats character declarations as signed. Use the –K compiler option to treat character declarations as unsigned. The behavior of C... more (see page 666)
Wide-character And Multi-character Constants (see page 666)	Wide-character types can be used to represent a character that does not fit into the storage space allocated for a **char** type. A wide character is stored in a two-byte space. A character constant preceded immediately by an L is a wide-character constant of data type wchar_t (defined in stddef.h). For example:

3.1.3.5.2.1.1.1 Character Constants

A character constant is one or more characters enclosed in single quotes, such as 'A', '+', or '\n'. In C, single-character constants have data type **int**. In C++, a character constant has type **char**. Multicharacter constants in both C and C++ have data type **int**.

To learn more about character constants, see the following topics.

- Three char types (see page 666)
- Escape sequences (see page 664)
- Wide-character and multi-character constants (see page 666)

Note: To compare sizes of character types, compile this as a C program and then as a C++ program.

```
#include <stdio.h>
#define CH 'x'       /* A CHARACTER CONSTANT */
void main(void) {
   char ch = 'x';    /* A char VARIABLE      */
   printf("\nSizeof int     = %d", sizeof(int) );
   printf("\nSizeof char    = %d", sizeof(char) );
   printf("\nSizeof ch      = %d", sizeof(ch) );
   printf("\nSizeof CH      = %d", sizeof(CH) );
   printf("\nSizeof wchar_t = %d", sizeof(wchar_t) );
}
```

Note: Sizes are in bytes.

Sizes of character types

Output when compiled as C program	Output when compiled as C++ program
Sizeof int = 4	Sizeof int = 4
Sizeof char = 1	Sizeof char = 1
Sizeof ch = 1	Sizeof ch = 1
Sizeof CH = 4	Sizeof CH = 1
Sizeof wchar_t = 2	Sizeof wchar_t = 2

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

3.1.3.5.2.1.1.2 Escape Sequences

The backslash character (\) is used to introduce an escape sequence, which allows the visual representation of certain nongraphic characters. For example, the constant **\n** is used to the single newline character.

A backslash is used with octal or hexadecimal numbers to represent the ASCII symbol or control code corresponding to that value; for example, '\03' for Ctrl-C or '\x3F' for the question mark. You can use any string of up to three octal or any number of hexadecimal numbers in an escape sequence, provided that the value is within legal range for data type **char** (0 to 0xff). Larger numbers generate the compiler error Numeric constant too large. For example, the octal number \777 is larger than the maximum value allowed (\377) and will generate an error. The first nonoctal or nonhexadecimal character encountered in an octal or hexadecimal escape sequence marks the end of the sequence.

Take this example.

```
printf("\x0072.1A Simple Operating System");
```

This is intended to be interpreted as \x007 and "2.1A Simple Operating System". However, the compiler treats it as the hexadecimal number \x0072 and the literal string "2.1A Simple Operating System".

To avoid such problems, rewrite your code like this:

```
printf("\x007" "2.1A Simple Operating System");
```

Ambiguities might also arise if an octal escape sequence is followed by a nonoctal digit. For example, because 8 and 9 are not legal octal digits, the constant \258 would be interpreted as a two-character constant made up of the characters \25 and 8.

The following table shows the available escape sequences.

Escape sequences

Note: You must use \\ to represent an ASCII backslash, as used in operating system paths.

Sequence	Value	Char	What it does
\a	0x07	BEL	Audible bell
\b	0x08	BS	Backspace
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Newline (linefeed)
\r	0x0D	CR	Carriage return
\t	0x09	HT	Tab (horizontal)
\v	0x0B	VT	Vertical tab
\\	0x5c	\	Backslash
\'	0x27	'	Single quote (apostrophe)
\"	0x22	"	Double quote
\?	0x3F	?	Question mark
\O		any	O=a string of up to three octal digits
\xH		any	H=a string of hex digits
\XH		any	H=a string of hex digits

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

3.1.3.5.2.1.1.3 The Three Char Types

One-character constants, such as 'A', '\t' and '007', are represented as **int** values. In this case, the low-order byte is sign extended into the high bit; that is, if the value is greater than 127 (base 10), the upper bit is set to -1 (=0xFF). This can be disabled by declaring that the default **char** type is **unsigned.**

The three character types, **char**, **signed char**, and **unsigned char,** require an 8-bit (one byte) storage. By default, the compiler treats character declarations as signed. Use the –K compiler option to treat character declarations as unsigned. The behavior of C programs is unaffected by the distinction between the three character types.

In a C++ program, a function can be overloaded with arguments of type **char**, **signed char**, or **unsigned char**. For example, the following function prototypes are valid and distinct:

```
void func(char ch);
void func(signed char ch);
void func(unsigned char ch);
```

If only one of the above prototypes exists, it will accept any of the three character types. For example, the following is acceptable:

```
void func(unsigned char ch);
void main(void)
{
  signed char ch = 'x';
  func(ch);
}
```

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

3.1.3.5.2.1.1.4 Wide-character And Multi-character Constants

Wide-character types can be used to represent a character that does not fit into the storage space allocated for a **char** type. A wide character is stored in a two-byte space. A character constant preceded immediately by an L is a wide-character constant of data type wchar_t (defined in stddef.h). For example:

```
wchar_t ch = L'A';
```

When wchar_t is used in a C program it is a type defined in stddef.h header file. In a C++ program, **wchar_t** is a keyword that can represent distinct codes for any element of the largest extended character set in any of the supported locales. In CodeGear C++, **wchar_t** is the same size, signedness, and alignment requirement as an unsigned short type.

A string preceded immediately by an L is a wide-character string. The memory allocation for a string is two bytes per character. For example:

```
wchar_t *str = L"ABCD";
```

Multi-character constants

The compiler also supports multi-character constants. Multi-character constants can consist of as many as four characters. For example, the constant, '\006\007\008\009' is valid only in a CodeGear C++ program. Multi-character constants are always 32-bit **int** values. The constants are not portable to other C++ compilers.

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

3.1.3.5.2.1.2 Constants

Constants are tokens representing fixed numeric or character values.

The compiler supports four classes of constants: integer, floating point, character (including strings), and enumeration.

Internal representation of numerical types shows how these types are represented internally.

The data type of a constant is deduced by the compiler using such clues as numeric value and the format used in the source code. The formal definition of a constant is shown in the following table.

Constants: Formal Definitions

constant: floating-constant integer-constant numeration-constant character-constant	nonzero-digit: one of 1 2 3 4 5 6 7 8 9
floating-constant: fractional-constant <exponent-part> <floating-suffix> digit-sequence exponent-part <floating-suffix>	octal-digit: one of 0 1 2 3 4 5 6 7
fractional-constant: <digit-sequence> **.** digit-sequence digit-sequence **.** a b c d e f	hexadecimal-digit: one of 0 1 2 3 4 5 6 7 8 9 A B C D E F

exponent-part: e <sign> digit-sequence E <sign> digit-sequence	integer-suffix: unsigned-suffix <long-suffix> long-suffix <unsigned-suffix>
sign: one of • -	unsigned-suffix: one of u U
digit-sequence: digit digit-sequence digit	long-suffix: one of l L
floating-suffix: one of f l F L	enumeration-constant: identifier
integer-constant: decimal-constant <integer-suffix> octal-constant <integer-suffix> hexadecimal-constant <integer-suffix>	character-constant c-char-sequence
decimal-constant: nonzero-digit decimal-constant digit	c-char-sequence: c-char c-char-sequence c-char
octal-constant: 0 octal-constant octal-digit	c-char: Any character in the source character set except the single-quote ('), backslash (\), or newline character escape-sequence.
hexadecimal-constant: 0 x hexadecimal-digit 0 X hexadecimal-digi hexadecimal-constant hexadecimal-digit	escape-sequence: one of the following \" \' \? \\ t \a \b \f \n \o \oo \ooo \r \t \v \Xh... \xh...

See Also

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

__int8 (see page 526)

3.1.3.5.2.1.3 Integer Constant Without L Or U

Decimal constants

0 to 32,767	**int**
32,768 to 2,147,483,647	**long**
2,147,483,648 to 4,294,967,295	**unsigned long**
> 4294967295	truncated

Octal constants

00 to 077777	**int**
010000 to 0177777	**unsigned int**
02000000 to 017777777777	**long**
020000000000 to 037777777777	**unsigned long**
> 037777777777	truncated

Hexadecimal constants

0x0000 to 0x7FFF	**int**
0x8000 to 0xFFFF	**unsigned int**
0x10000 to 0x7FFFFFFF	**long**
0x80000000 to 0xFFFFFFFF	**unsigned long**
>0xFFFFFFFF	truncated

See Also

Constants (see page 667)

Integer Constants (see page 674)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

__int8 (see page 526)

3.1.3.5.2.1.4 Constant Expressions

This section contains Constant Expression topics.

Topics

Name	Description
Constant Expressions (see page 670)	A constant expression is an expression that always evaluates to a constant (it is evaluated at compile-time and it must evaluate to a constant that is in the range of representable values for its type). Constant expressions are evaluated just as regular expressions are. You can use a constant expression anywhere that a constant is legal. The syntax for constant expressions is:

3.1.3.5.2.1.4.1 Constant Expressions

A constant expression is an expression that always evaluates to a constant (it is evaluated at compile-time and it must evaluate to a constant that is in the range of representable values for its type). Constant expressions are evaluated just as regular expressions are. You can use a constant expression anywhere that a constant is legal. The syntax for constant expressions is:

```
constant-expression:
Conditional-expression
```

Constant expressions cannot contain any of the following operators, unless the operators are contained within the operand of a **sizeof** operator:

- Assignment
- Comma
- Decrement
- Function call
- Increment

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

3.1.3.5.2.1.5 Constants and Internal Representation

This section contains Constants and Internal Representation topics.

Topics

Name	Description
Constants And Internal Representation (see page 671)	ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation-specific and usually derive from the architecture of the host computer. For CodeGear C++, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 80x86 and the Pentium family of microprocessors governs the choices of internal representations for the various data types. The following tables list the sizes and resulting ranges of the data types. Internal representation of numerical types shows how these types are represented internally. **32-bit data types, sizes, and ranges**

3.1.3.5.2.1.5.1 Constants And Internal Representation

ANSI C acknowledges that the size and numeric range of the basic data types (and their various permutations) are implementation-specific and usually derive from the architecture of the host computer. For CodeGear C++, the target platform is the IBM PC family (and compatibles), so the architecture of the Intel 80x86 and the Pentium family of microprocessors governs the choices of internal representations for the various data types.

The following tables list the sizes and resulting ranges of the data types. Internal representation of numerical types shows how these types are represented internally.

32-bit data types, sizes, and ranges

Type	Size (bits)	Range	Sample applications
unsigned char	8	0 <= X <= 255	Small numbers and full PC character set
char	8	-128 <= X <= 127	Very small numbers and ASCII characters
short int	16	-32,768 <= X <= 32,767	Counting, small numbers, loop control
unsigned int	32	0 <= X <= 4,294,967,295	Large numbers and loops
int	32	-2,147,483,648 <= X <= 2,147,483,647	Counting, small numbers, loop control
unsigned long	32	0 <= X <= 4,294,967,295	Astronomical distances
enum	32	-2,147,483,648 <= X <= 2,147,483,647	Ordered sets of values
long	32	-2,147,483,648 <= X <= 2,147,483,647	Large numbers, populations
float	32	1.18 (10^-38 < |X| < 3.40 (10^38	Scientific (7-digit) precision)
double	64	2.23 (10^-308 < |X| < 1.79 (10^308	Scientific (15-digit precision)
long double	80	3.37 (10^-4932 < |X| < 1.18 (10^4932	Financial (18-digit precision)

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

__int8 (see page 526)

The Three Char Types (see page 666)

Internal Representation Of Numerical Types (see page 676)

3.1.3.5.2.1.6 Floating Point Constants

A floating-point constant consists of:

- Decimal integer
- Decimal point
- Decimal fraction
- e or E and a signed integer exponent (optional)
- Type suffix: f or F or l or L (optional)

You can omit either the decimal integer or the decimal fraction (but not both). You can omit either the decimal point or the letter e (or E) and the signed integer exponent (but not both). These rules allow for conventional and scientific (exponent) notations.

Negative floating constants are taken as positive constants with the unary operator minus (-) prefixed.

Here are some examples:

Constant	Value
23.45e6	23.45 (10^6
.0	0
0.	0.0
1.	1.0
-1.23	-1.23
2e-5	2.0 (10^-5
3E+10	3.0 (10^10
.09E34	0.09 (10^34

In the absence of any suffixes, floating-point constants are of type **double.** However, you can coerce a floating constant to be of type **float** by adding an f or F suffix to the constant. Similarly, the suffix l or L forces the constant to be data type **long double**. The table below shows the ranges available for **float**, **double**, and **long double**.

Floating-point constant sizes and ranges

Type	Size (bits)	Range
float	32	3.4 (10^-38 to 3.4 (10^38
double	64	1.7 (10^-308 to 1.7 (10^308
long double	80	3.4 (10^-4932 to 1.1 (10^4932

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

3.1.3.5.2.1.7 Enumeration Constants

This section contains Enumeration Constant topics.

Topics

Name	Description
Enumeration Constants (see page 673)	Enumeration constants are identifiers defined in **enum** type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the **enum** declaration. Negative initializers are allowed. See Enumerations and enum (keyword) for a detailed look at **enum** declarations. The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional initializers. In this example,

3.1.3.5.2.1.7.1 Enumeration Constants

Enumeration constants are identifiers defined in **enum** type declarations. The identifiers are usually chosen as mnemonics to assist legibility. Enumeration constants are integer data types. They can be used in any expression where integer constants are valid. The identifiers used must be unique within the scope of the **enum** declaration. Negative initializers are allowed. See Enumerations and enum (keyword) for a detailed look at **enum** declarations.

The values acquired by enumeration constants depend on the format of the enumeration declaration and the presence of optional initializers. In this example,

```
enum team { giants, cubs, dodgers };
```

giants, cubs, and dodgers are enumeration constants of type team that can be assigned to any variables of type team or to any other variable of integer type. The values acquired by the enumeration constants are

```
giants = 0, cubs = 1, dodgers = 2
```

in the absence of explicit initializers. In the following example,

```
enum team { giants, cubs=3, dodgers = giants + 1 };
```

the constants are set as follows:

```
giants = 0, cubs = 3, dodgers = 1
```

The constant values need not be unique:

```
enum team { giants, cubs = 1, dodgers = cubs - 1 };
```

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

3.1.3.5.2.1.8 Integer Constants

Integer constants can be decimal (base 10), octal (base 8) or hexadecimal (base 16). In the absence of any overriding suffixes, the data type of an integer constant is derived from its value, as shown in Integer constants without L or U.. Note that the rules vary between decimal and nondecimal constants.

Decimal

Decimal constants from 0 to 4,294,967,295 are allowed. Constants exceeding this limit are truncated. Decimal constants must not use an initial zero. An integer constant that has an initial zero is interpreted as an octal constant. Thus,

```
int i = 10;  /*decimal 10 */
int i = 010; /*decimal 8 */
int i = 0;   /*decimal 0 = octal 0 */
```

Octal

All constants with an initial zero are taken to be octal. If an octal constant contains the illegal digits 8 or 9, an error is reported. Octal constants exceeding 037777777777 are truncated.

Hexadecimal

All constants starting with 0x (or 0X) are taken to be hexadecimal. Hexadecimal constants exceeding 0xFFFFFFFF are truncated.

long and unsigned suffixes

The suffix L (or l) attached to any constant forces the constant to be represented as a **long**. Similarly, the suffix U (or u) forces the constant to be **unsigned**. It is **unsigned long** if the value of the number itself is greater than decimal 65,535, regardless of

which base is used. You can use both L and U suffixes on the same constant in any order or case: ul, lu, UL, and so on.

The data type of a constant in the absence of any suffix (U, u, L, or l) is the first of the following types that can accommodate its value:

Decimal	**int, long int, unsigned long int**
Octal	**int, unsigned int, long int, unsigned long int**
Hexadecimal	**int, unsigned int, long int, unsigned long int**

If the constant has a U or u suffix, its data type will be the first of **unsigned int, unsigned long int** that can accommodate its value.

If the constant has an L or l suffix, its data type will be the first of **long int, unsigned long int** that can accommodate its value.

If the constant has both u and l suffixes, (ul, lu, Ul, lU, uL, Lu, LU or UL), its data type will be **unsigned long int**.

Integer constants without L or U summarizes the representations of integer constants in all three bases. The data types indicated assume no overriding L or U suffix has been used.

See Also

Constants (see page 667)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

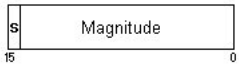
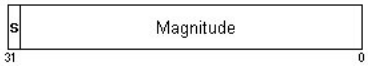
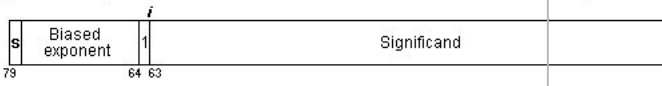
Constant Expressions (see page 670)

__int8 (see page 526)

3.1.3.5.2.1.9 Internal Representation of Numerical Types

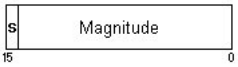
This section contains Internal Representation of Numerical Type topics.

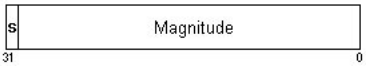
Topics

Name	Description
Internal Representation Of Numerical Types (see page 676)	**32-bit integers** **short int**  (2's complement) **int, long int**  (2's complement) **Floating-point types, always** **float**  **double**  **long double** 

3.1.3.5.2.1.9.1 Internal Representation Of Numerical Types

32-bit integers

short int  (2's complement)

int, long int  (2's complement)

Floating-point types, always

float 

double 

long double 

s	=	Sign bit (0 = positive, 1 = negative)	Exponent bias (normalized values):
i	=	Position of implicit binary point	**float:** 127 (7FH)
1	=	Integer bit of significance:	**double:** 1,023 (3FFH)
		Stored in **long double** Implicit in **float**, **double**	**long double:** 16,383 (3FFFH)

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

String Constants (see page 677)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Constant Expressions (see page 670)

3.1.3.5.2.1.10 String Constants

This section contains String Constant topics.

Topics

Name	Description
String Constants (see page 677)	String constants, also known as string literals, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type array-of- **const char** and storage class **static**, written as a sequence of any number of characters surrounded by double quotes:

3.1.3.5.2.1.10.1 String Constants

String constants, also known as string literals, form a special category of constants used to handle fixed sequences of characters. A string literal is of data type array-of- **const char** and storage class **static**, written as a sequence of any number of characters surrounded by double quotes:

```
"This is literally a string!"
```

The null (empty) string is written "".

The characters inside the double quotes can include escape sequences. This code, for example:

```
"\t\t\"Name\"\\\tAddress\n\n"
```

prints like this:

```
"Name"\        Address
```

"Name" is preceded by two tabs; Address is preceded by one tab. The line is followed by two new lines. The \" provides interior double quotes.

If you compile with the -**A** option for ANSI compatibility, the escape character sequence "\\", is translated to "\" by the compiler.

A literal string is stored internally as the given sequence of characters plus a final null character ('\0'). A null string is stored as a single '\0' character.

Adjacent string literals separated only by whitespace are concatenated during the parsing phase. In the following example,

```
#include <stdio.h>
```

```
int main() {
   char    *p;
   p = "This is an example of how the compiler "   " will\nconcatenate very long strings for
you"   " automatically, \nresulting in nicer" " looking programs.";
   printf(p);
   return(0);
}
```

The output of the program is

```
This is an example of how the compiler will
concatenate very long strings for you automatically,
resulting in nicer looking programs.
```

You can also use the backslash (\) as a continuation character to extend a string constant across line boundaries:

```
puts("This is really \
a one-line string");
```

See Also

Constants (see page 667)

Integer Constants (see page 674)

Integer Constant Without L Or U (see page 669)

Floating Point Constants (see page 672)

Character Constants (see page 663)

The Three Char Types (see page 666)

Escape Sequences (see page 664)

Wide-character And Multi-character Constants (see page 666)

Enumeration Constants (see page 673)

Constants And Internal Representation (see page 671)

Internal Representation Of Numerical Types (see page 676)

Constant Expressions (see page 670)

3.1.3.5.2.2 Tokens

Tokens are word-like units recognized by a language. The compiler recognizes six classes of tokens.

Here is the formal definition of a token:

- keyword
- identifier
- constant
- string-literal
- operator
- punctuator (also known as separators)

As the source code is scanned, tokens are extracted in such a way that the longest possible token from the character sequence is selected. For example, external would be parsed as a single identifier, rather than as the keyword **extern** followed by the identifier al.

See Token Pasting with ## for a description of token pasting.

See Also

Lexical Elements (see page 661)

Whitespace (see page 686)

3.1.3.5.2.3 Identifiers Overview

This section contains Identifier topics.

Topics

Name	Description
Identifiers (see page 679)	Here is the formal definition of an identifier: identifier: • nondigit • identifier nondigit • identifier digit nondigit: one of • a b c d e f g h i j k l m n o p q r s t u v w x y z _ • A B C D E F G H I J K L M N O P Q R S T U V W X Y Z digit: one of • 0 1 2 3 4 • ef5 6 7 8 9 **Naming and length restrictions** Identifiers are arbitrary names of any length given to classes, objects, functions, variables,... more (see page 679)

3.1.3.5.2.3.1 Identifiers

Here is the formal definition of an identifier:

identifier:

- nondigit
- identifier nondigit
- identifier digit

nondigit: one of

- a b c d e f g h i j k l m n o p q r s t u v w x y z _
- A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

digit: one of

- 0 1 2 3 4
- ef5 6 7 8 9

Naming and length restrictions

Identifiers are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types, and so on. (Identifiers can contain the letters a to z and A to Z, the underscore character "_", and the digits 0 to 9.) There are only two restrictions:

- The first character must be a letter or an underscore.

- By default, the compiler recognizes only the first 250 characters as significant. The number of significant characters can be reduced by menu and command-line options, but not increased. To change the significant character length, use the spin control in Project|Options|Advanced Compiler|Source|Identifier Length.

Case sensitivity

Identifiers in C and C++ are case sensitive, so that Sum, sum and suM are distinct identifiers.

Global identifiers imported from other modules follow the same naming and significance rules as normal identifiers. However, you have the option of suspending case sensitivity to allow compatibility when linking with case-insensitive languages. With the case-insensitive option, the globals Sum and sum are considered identical, resulting in a possible. "Duplicate symbol" warning during linking.

An exception to these rules is that identifiers of type __**pascal** are always converted to all uppercase for linking purposes.

Uniqueness and scope

Although identifier names are arbitrary (within the rules stated), errors result if the same name is used for more than one identifier within the same scope and sharing the same name space. Duplicate names are legal for different name spaces regardless of scope rules.

3.1.3.5.2.4 Keywords Overview

This section contains Keyword topics.

Topics

Name	Description
Keywords (see page 680)	Keywords are words reserved for special purposes and must not be used as normal identifier names. If you use non-ANSI keywords in a program and you want the program to be ANSI compliant, always use the non-ANSI keyword versions that are prefixed with double underscores. Some keywords have a version prefixed with only one underscore; these keywords are provided to facilitate porting code developed with other compilers. For ANSI-specified keywords there is only one version. **Note:** Note that the keywords __try and **try** are an exception to the discussion above. The keyword **try** is required to match the **catch** keyword... more (see page 680)
Keyword Extensions (see page 681)	CodeGear C++Builder provides a number of keywords that are not part of the ANSI Standard. Please see the Help table of contents for a complete categorical and alphabetical listing of Library Routines.
C++-Specific Keywords (see page 681)	A number of keywords are specific to C++ and are not available if you are writing a program in C only. Please see the Help table of contents for a complete categorical and alphabetical listing of these and other keywords.
Table Of CodeGear C++ Register Pseudovariables (see page 681)	This table lists all pseudovariables.

3.1.3.5.2.4.1 Keywords

Keywords are words reserved for special purposes and must not be used as normal identifier names.

If you use non-ANSI keywords in a program and you want the program to be ANSI compliant, always use the non-ANSI keyword versions that are prefixed with double underscores. Some keywords have a version prefixed with only one underscore; these keywords are provided to facilitate porting code developed with other compilers. For ANSI-specified keywords there is only one version.

Note: Note that the keywords __try

and **try** are an exception to the discussion above. The keyword **try** is required to match the **catch** keyword in the C++ exception-handling mechanism. **try** cannot be substituted by __**try.** The keyword __**try** can only be used to match the __**except** or __**finally** keywords. See the discussions on C++ exception handling and C-based structured exceptions under Win32 for more information. Please see the Help table of contents for a complete categorical and alphabetical listing of keywords.

See Also

Alphabetical Listing of Keywords (see page 513)

Keywords By Category (see page 579)

3.1.3.5.2.4.2 Keyword Extensions

CodeGear C++Builder provides a number of keywords that are not part of the ANSI Standard.

Please see the Help table of contents for a complete categorical and alphabetical listing of Library Routines.

See Also

C++Builder Keyword Extensions (see page 580)

Keywords (see page 513)

Keywords By Category (see page 579)

3.1.3.5.2.4.3 C++-Specific Keywords

A number of keywords are specific to C++ and are not available if you are writing a program in C only. Please see the Help table of contents for a complete categorical and alphabetical listing of these and other keywords.

See Also

Alphabetical Listing of Keywords (see page 513)

Keywords By Category (see page 579)

3.1.3.5.2.4.4 Table Of CodeGear C++ Register Pseudovariables

This table lists all pseudovariables.

_AH	**_CL**	**_EAX**	**_ESP**
_AL	**_CS**	**_EBP**	**_FLAGS**
_AX	**_CX**	**_EBX**	**_FS**
_BH	**_DH**	**_ECX**	**_GS**
_BL	**_DI**	**_EDI**	**_SI**
_BP	**_DL**	**_EDX**	**_SP**
_BX	**_DS**	**_ES**	**_SS**
_CH	**_DX**	**_ESI**	

All but the _FLAGS and _EFLAGS register pseudovariables are associated with the general purpose, segment, address, and special purpose registers.

Use register pseudovariables anywhere that you can use an integer variable to directly access the corresponding 80x86 register.

The flags registers contain information about the state of the 80x86 and the results of recent instructions.

3.1.3.5.2.5 Punctuators Overview

This section contains Punctuator topics.

Topics

Name	Description
Punctuators (see page 682)	The C++ punctuators (also known as separators) are: • [ ] • ( ) • { } • , • ; • : • ... • = • # Most of these punctuators also function as operators. **Brackets** Open and close brackets indicate single and multidimensional array subscripts:

3.1.3.5.2.5.1 Punctuators

The C++ punctuators (also known as separators) are:

- [ ]
- ( )
- { }
- ,
- ;
- :
- ...
- =
- #

Most of these punctuators also function as operators.

Brackets

Open and close brackets indicate single and multidimensional array subscripts:

```
char ch, str[] = "Stan";
int mat[3][4];           /* 3 x 4 matrix */
ch = str[3];             /* 4th element */
  .
  .
  .
```

Parentheses

Open and close parentheses **( )** are used to group expressions, isolate conditional expressions, and indicate function calls and function parameters:

```
d = c * (a + b);/* override normal precedence */
if (d == z) ++x;/* essential with conditional statement */
```

```
func();/* function call, no args */
int (*fptr)();/* function pointer declaration */
fptr = func;/* no () means func pointer */
void func2(int n);/* function declaration with parameters */
```

Parentheses are recommended in macro definitions to avoid potential precedence problems during expansion:

```
#define CUBE(x) ((x) * (x) * (x))
```

The use of parentheses to alter the normal operator precedence and associativity rules is covered in Expressions.

Braces

Open and close braces **{ }** indicate the start and end of a compound statement:

```
if (d == z)
{
   ++x;
   func();
}
```

The closing brace serves as a terminator for the compound statement, so a ; (semicolon) is not required after the }, except in structure or class declarations. Often, the semicolon is illegal, as in

```
if (statement)
   {};                  /* illegal semicolon */
else
```

Comma

The comma (**,**) separates the elements of a function argument list:

```
void func(int n, float f, char ch);
```

The comma is also used as an operator in comma expressions. Mixing the two uses of comma is legal, but you must use parentheses to distinguish them. Note that (exp1, exp2) evaluates both but is equal to the second:

```
func(i, j);                              /* call func with two args */
func((exp1, exp2), (exp3, exp4, exp5));  /* also calls func with two args! */
```

Semicolon

The semicolon (**;**) is a statement terminator. Any legal C or C++ expression (including the empty expression) followed by a semicolon is interpreted as a statement, known as an expression statement. The expression is evaluated and its value is discarded. If the expression statement has no side effects, the compiler might ignore it.

```
a + b;    /* maybe evaluate a + b, but discard value */
++a;      /* side effect on a, but discard value of ++a */
;         /* empty expression = null statement */
```

Semicolons are often used to create an empty statement:

```
for (i = 0; i < n; i++)
{
   ;
}
```

Colon

Use the colon (**:**) to indicate a labeled statement:

```
start:    x=0;
...
goto start;
```

Labels are discussed in Labeled statements.

Ellipsis

The ellipsis

(**...**) is three successive periods with no intervening whitespace. Ellipses are used in the formal argument lists of function prototypes to indicate a variable number of arguments, or arguments with varying types:

```
void func(int n, char ch,...);
```

This declaration indicates that func will be defined in such a way that calls must have at least two arguments, an **int** and a **char**, but can also have any number of additional arguments.

In C++, you can omit the comma before the ellipsis.

Asterisk (pointer declaration)

The asterisk (*****) in a variable declaration denotes the creation of a pointer to a type:

```
char *char_ptr;  /* a pointer to char is declared */
```

Pointers with multiple levels of indirection can be declared by indicating a pertinent number of asterisks:

```
int **int_ptr;         /* a pointer to an integer array */
double ***double_ptr;  /* a pointer to a matrix of doubles */
```

You can also use the asterisk as an operator to either dereference a pointer or as the multiplication operator:

```
i = *int_ptr;
a = b * 3.14;
```

Equal sign (initializer)

The equal sign (**=**) separates variable declarations from initialization lists:

```
char array[5] = { 1, 2, 3, 4, 5 };
int  x = 5;
```

In C++, declarations of any type can appear (with some restrictions) at any point within the code. In a C function, no code can precede any variable declarations.

In a C++ function argument list, the equal sign indicates the default value for a parameter:

```
int f(int i = 0) { ... }  /* Parameter i has default value of zero */
```

The equal sign is also used as the assignment operator in expressions:

```
int a, b, c;
a = b + c;
float *ptr = (float *) malloc(sizeof(float) * 100);
```

Pound sign (preprocessor directive)

The pound sign (#) indicates a preprocessor directive when it occurs as the first nonwhitespace character on a line. It signifies a compiler action, not necessarily associated with code generation. See Preprocessor directives for more on the preprocessor directives.

and ## (double pound signs) are also used as operators to perform token replacement and merging during the preprocessor scanning phase. See Token pasting.

3.1.3.5.3 Whitespace Overview

This section contains Whitespace Overview topics.

Topics

Name	Description
Comments (see page 685)	Comments are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing. There are two ways to delineate comments: the C method and the C++ method. The compiler supports both methods, and provides an additional, optional extension permitting nested comments. If you are not compiling for ANSI compatibility, you can use any of these kinds of comments in both C and C++ programs. You should also follow the guidelines on the use of whitespace and delimiters in comments discussed later in this topic to avoid other... more (see page 685)
Whitespace (see page 686)	Whitespace is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, the two sequences

3.1.3.5.3.1 Comments

Comments are pieces of text used to annotate a program. Comments are for the programmer's use only; they are stripped from the source text before parsing.

There are two ways to delineate comments: the C method and the C++ method. The compiler supports both methods, and provides an additional, optional extension permitting nested comments. If you are not compiling for ANSI compatibility, you can use any of these kinds of comments in both C and C++ programs.

You should also follow the guidelines on the use of whitespace and delimiters in comments discussed later in this topic to avoid other portability problems.

C comments

A C comment is any sequence of characters placed after the symbol pair /*. The comment terminates at the first occurrence of the pair ***/** following the initial /*. The entire sequence, including the four comment-delimiter symbols, is replaced by one space after macro expansion. Note that some C implementations remove comments without space replacements.

The compiler does not support the nonportable token pasting strategy using /**/. Token pasting is performed with the ANSI-specified pair ##, as follows:

```
#define VAR(i,j)  (i/**/j)    /* won't work */
#define VAR(i,j)  (i##j)      /* OK */
#define VAR(i,j)  (i ## j)    /* Also OK */
```

The compiler parses the declaration,

```
int /* declaration */ i /* counter */;
```

as these three tokens:

```
int   i;
```

See Token Pasting with ## for a description of token pasting.

C++ comments

C++ allows a single-line comment using two adjacent slashes (//). The comment can start in any position, and extends until the next new line:

```
class X {  // this is a comment
... };
```

You can also use // to create comments in C code. This feature is specific to the CodeGear C++ compiler and is generally not portable.

Nested comments

ANSI C doesn't allow nested comments. The attempt to comment out a line

```
/*  int /* declaration */ i /* counter */; */
```

fails, because the scope of the first **/*** ends at the first */. This gives

```
i ; */
```

which would generate a syntax error.

To allow nested comments, check Project|Options|Advanced Compiler|Source|Nested Comments.

Delimiters and whitespace

In rare cases, some whitespace before /* and //, and after */, although not syntactically mandatory, can avoid portability problems. For example, this C++ code:

```
int i = j//* divide by k*/k;
+m;
```

parses as int i = j +m; not as

```
int i = j/k;
+m;
```

as expected under the C convention. The more legible

```
int i = j/ /* divide by k*/ k;
+m;
```

avoids this problem.

See Also

Lexical Elements (see page 661)

Whitespace (see page 686)

Tokens (see page 678)

3.1.3.5.3.2 Whitespace

Whitespace is the collective name given to spaces (blanks), horizontal and vertical tabs, newline characters, and comments. Whitespace can serve to indicate where tokens start and end, but beyond this function, any surplus whitespace is discarded. For example, the two sequences

```
int i; float f;
```

and

```
int i;
   float f;
```

are lexically equivalent and parse identically to give the six tokens:

- **int**
- i
- **;**
- **float**
- **f**
- **;**

The ASCII characters representing whitespace can occur within literal strings, in which case they are protected from the normal parsing process (they remain as part of the string). For example,

```
char name[] = "CodeGear Corporation";
```

parses to seven tokens, including the single literal-string token "CodeGear Corporation"

**Line splicing with **

A special case occurs if the final newline character encountered is preceded by a backslash (\). The backslash and new line are both discarded, allowing two physical lines of text to be treated as one unit.

```
"CodeGear \
Corporation"
```

is parsed as "CodeGear Corporationl" (see String constants for more information).

See Also

Lexical Elements (see page 661)

Tokens (see page 678)

3.1.3.6 The Preprocessor

This section contains Preprocessor topics.

Topics

Name	Description
Conditional Compilation Overview (see page 687)	This section contains Conditional Compilation Overview topics.
Defining And Undefining Macros (see page 689)	This section contains Defining And Undefining Macro topics.
The #error Control Directive (see page 691)	This section contains #error Control Directive topics.
File Inclusion With #include (see page 692)	This section contains File Inclusion With #include topics.
The #line Control Directive (see page 693)	This section contains #line Control Directive topics.
Macros With Parameters Overview (see page 694)	This section contains Macros With Parameters Overview topics.
Pragma Directives Overview (see page 696)	This section contains Pragma Directives Overview topics.
Predefined Macros Overview (see page 707)	This section contains Predefined Macros Overview topics.
Preprocessor Directives (see page 710)	This section contains C++ Preprocessor Directive topics.

3.1.3.6.1 Conditional Compilation Overview

This section contains Conditional Compilation Overview topics.

Topics

Name	Description
Conditional Compilation (see page 687)	The compiler supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those beginning with # (except the **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif** directives), as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.
#if, #elif, #else, And #endif (see page 688)	**Syntax**
#ifdef And #ifndef (see page 688)	**Syntax**
Defined (see page 689)	**Syntax**

3.1.3.6.1.1 Conditional Compilation

The compiler supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those beginning with # (except the **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif** directives), as well as any lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

3.1.3.6.1.2 #if, #elif, #else, And #endif

Syntax

```
#if constant-expression-1
<section-1>
<#elif constant-expression-2 newline section-2>
  .
  .
  .
<#elif constant-expression-n newline section-n>
<#else <newline> final-section>
#endif
```

Description

The compiler supports conditional compilation by replacing the appropriate source-code lines with a blank line. The lines thus ignored are those lines that are not to be compiled as a result of the directives. All conditional compilation directives must be completed in the source or include file in which they are begun.

The conditional directives **#if**, **#elif**, **#else**, and **#endif** work like the normal C conditional operators. If the constant-expression-1 (subject to macro expansion) evaluates to nonzero (true), the lines of code (possibly empty) represented by section-1, whether preprocessor command lines or normal source lines, are preprocessed and, as appropriate, passed to the compiler. Otherwise, if constant-expression-1 evaluates to zero (false), section-1 is ignored (no macro expansion and no compilation).

In the true case, after section-1 has been preprocessed, control passes to the matching **#endif** (which ends this conditional sequence) and continues with next-section. In the false case, control passes to the next **#elif** line (if any) where constant-expression-2 is evaluated. If true, section-2 is processed, after which control moves on to the matching **#endif**. Otherwise, if constant-expression-2 is false, control passes to the next **#elif**, and so on, until either **#else** or **#endif** is reached. The optional **#else** is used as an alternative condition for which all previous tests have proved false. The **#endif** ends the conditional sequence.

The processed section can contain further conditional clauses, nested to any depth; each **#if** must be matched with a closing **#endif**.

The net result of the preceding scenario is that only one section (possibly empty) is passed on for further processing. The bypassed sections are relevant only for keeping track of any nested conditionals, so that each **#if** can be matched with its correct **#endif**.

The constant expressions to be tested must evaluate to a constant integral value.

See Also

defined (see page 689)

#ifdef And #ifndef (see page 688)

3.1.3.6.1.3 #ifdef And #ifndef

Syntax

```
#ifdef identifier
#ifndef identifier
```

Description

The **#ifdef** and **#ifndef** conditional directives let you test whether an identifier is currently defined or not; that is, whether a previous **#define** command has been processed for that identifier and is still in force. The line

```
#ifdef identifier
```

has exactly the same effect as

```
#if 1
```

if identifier is currently defined, and the same effect as

```
#if 0
```

if identifier is currently undefined.

#ifndef tests true for the "not-defined" condition, so the line

```
#ifndef identifier
```

has exactly the same effect as

```
#if 0
```

if identifier is currently defined, and the same effect as

```
#if 1
```

if identifier is currently undefined.

The syntax thereafter follows that of the **#if**, **#elif**, **#else**, and **#endif**.

An identifier defined as NULL is considered to be defined.

3.1.3.6.1.4 Defined

Syntax

```
#if defined[(] <identifier> [)]
#elif defined[(] <identifier> [)]
```

Description

Use the **defined** operator to test if an identifier was previously defined using **#define**. The **defined** operator is only valid in **#if** and **#elif** expressions.

Defined evaluates to 1 (true) if a previously defined symbol has not been undefined (using **#undef**); otherwise, it evaluates to 0 (false).

Defined performs the same function as **#ifdef**.

```
#if defined(mysym)
```

is the same as

```
#ifdef mysym
```

The advantage is that you can use defined repeatedly in a complex expression following the **#if** directive; for example,

```
#if defined(mysym) && !defined(yoursym)
```

See Also

#ifdef And #ifndef (see page 688)

3.1.3.6.2 Defining And Undefining Macros

This section contains Defining And Undefining Macro topics.

Topics

Name	Description
Using The -D And -U Command-line Options (see page 690)	Identifiers can be defined and undefined using the command-line compiler options -D and -U. The command line
Keywords And Protected Words In Macros (see page 690)	It is legal but ill-advised to use C++ keywords as macro identifiers:
#define (see page 690)	**Syntax**
#undef (see page 691)	**Syntax**

3.1.3.6.2.1 Using The -D And -U Command-line Options

Identifiers can be defined and undefined using the command-line compiler options -D and -U.

The command line

```
BCC32 -Ddebug=1; paradox=0; X -Umysym myprog.cbc++ -Ddebug=1:paradox=0:X -Umysym myprog.c
bc++ -Ddebug=1:paradox=0:X -Umysym myprog.c
```

is equivalent to placing

```
#define debug 1
#define paradox 0
```

in the program.

See Also

#define (see page 690)

3.1.3.6.2.2 Keywords And Protected Words In Macros

It is legal but ill-advised to use C++ keywords as macro identifiers:

```
#define int long    /* legal but probably catastrophic */
#define INT long    /* legal and possibly useful */
```

The following predefined global identifiers cannot appear immediately following a **#define** or **#undef** directive:

- **__DATE__ __FILE__ __LINE__**
- **__STDC__ __TIME__**

3.1.3.6.2.3 #define

Syntax

```
#define macro_identifier <token_sequence>
```

Description

The **#define** directive defines a macro. Macros provide a mechanism for token replacement with or without a set of formal, function-like parameters.

Each occurrence of **macro_identifier** in your source code following this control line will be replaced in the original position with the possibly empty **token_sequence** (there are some exceptions, which are noted later). Such replacements are known as macro expansions. The token sequence is sometimes called the body of the macro.

An empty token sequence results in the removal of each affected macro identifier from the source code.

After each individual macro expansion, a further scan is made of the newly expanded text. This allows for the possibility of nested macros: The expanded text can contain macro identifiers that are subject to replacement. However, if the macro expands into what looks like a preprocessing directive, the directive will not be recognized by the preprocessor. There are these

restrictions to macro expansion:

- Any occurrences of the macro identifier found within literal strings, character constants, or comments in the source code are not expanded.
- A macro won't be expanded during its own expansion (so **#define** A A won't expand indefinitely).

Example

```
#define HI "Have a nice day!"
#define empty
```

See Also

Keywords And Protected Words In Macros (see page 690)

Macros With Parameters (see page 694)

#undef (see page 691)

Using The -D And -U Command-line Options (see page 690)

3.1.3.6.2.4 #undef

Syntax

```
#undef macro_identifier
```

Description

You can undefine a macro using the **#undef** directive. **#undef** detaches any previous token sequence from the macro identifier; the macro definition has been forgotten, and the macro identifier is undefined. No macro expansion occurs within **#undef** lines.

The state of being defined or undefined turns out to be an important property of an identifier, regardless of the actual definition. The **#ifdef** and **#ifndef** conditional directives, used to test whether any identifier is currently defined or not, offer a flexible mechanism for controlling many aspects of a compilation.

After a macro identifier has been undefined, it can be redefined with **#define**, using the same or a different token sequence.

Attempting to redefine an already defined macro identifier will result in a warning unless the new definition is exactly the same token-by-token definition as the existing one. The preferred strategy where definitions might exist in other header files is as follows:

```
#ifndef BLOCK_SIZE
  #define BLOCK_SIZE 512
```

The middle line is bypassed if BLOCK_SIZE is currently defined; if BLOCK_SIZE is not currently defined, the middle line is invoked to define it.

No semicolon (;) is needed to terminate a preprocessor directive. Any character found in the token sequence, including semicolons, will appear in the macro expansion. The token sequence terminates at the first non-backslashed new line encountered. Any sequence of whitespace, including comments in the token sequence, is replaced with a single-space character.

See Also

#define (see page 690)

Preprocessor Directives (see page 710)

3.1.3.6.3 The #error Control Directive

This section contains #error Control Directive topics.

Topics

Name	Description
#error (see page 692)	**Syntax**

3.1.3.6.3.1 #error

Syntax

```
#error errmsg
```

Description

The **#error** directive generates the message:

```
Error: filename line# : Error directive: errmsg
```

This directive is usually embedded in a preprocessor conditional statement that catches some undesired compile-time condition. In the normal case, that condition will be false. If the condition is true, you want the compiler to print an error message and stop the compile. You do this by putting an **#error** directive within a conditional statement that is true for the undesired case.

3.1.3.6.4 File Inclusion With #include

This section contains File Inclusion With #include topics.

Topics

Name	Description
#include (see page 692)	**Syntax**
Header File Search With <header_name> (see page 693)	The <header_name> version specifies a standard include file; the search is made successively in each of the include directories in the order they are defined. If the file is not located in any of the default directories, an error message is issued.
Header File Search With "header_name" (see page 693)	The "header_name" version specifies a user-supplied include file; the file is searched in the following order: 1. The same directory of the file that contains the **#include** statement 2. The directories of files that include (**#include**) that file 3. The current directory 4. The path specified by the /I compiler option

3.1.3.6.4.1 #include

Syntax

```
#include <header_name>
#include "header_name"
#include <macro_definition>
```

Description

The **#include** directive pulls in other named files, known as include files, header files, or headers, into the source code. The syntax has three versions:

- The first and second versions imply that no macro expansion will be attempted; in other words, header_name is never scanned for macro identifiers. header_name must be a valid file name with an extension (traditionally .h for header) and optional path name and path delimiters.
- The third version assumes that neither **<** nor " appears as the first non-whitespace character following **#include**; further, it assumes a macro definition exists that will expand the macro identifier into a valid delimited header name with either of the <header_name> or "header_name" formats.

The preprocessor removes the **#include** line and conceptually replaces it with the entire text of the header file at that point in the source code. The source code itself is not changed, but the compiler "sees" the enlarged text. The placement of the **#include** can therefore influence the scope and duration of any identifiers in the included file.

If you place an explicit path in the header_name, only that directory will be searched.

The difference between the <header_name> and "header_name" formats lies in the searching algorithm employed in trying to locate the include file.

See Also

Header File Search with <header_name> (see page 693)

Header File Search with “header_name” (see page 693)

3.1.3.6.4.2 Header File Search With <header_name>

The <header_name> version specifies a standard include file; the search is made successively in each of the include directories in the order they are defined. If the file is not located in any of the default directories, an error message is issued.

See Also

Header File Search With "header_name" (see page 693)

3.1.3.6.4.3 Header File Search With "header_name"

The "header_name" version specifies a user-supplied include file; the file is searched in the following order:

1. The same directory of the file that contains the **#include** statement
2. The directories of files that include (**#include**) that file
3. The current directory
4. The path specified by the /I compiler option

See Also

Header File Search With <header_name> (see page 693)

3.1.3.6.5 The #line Control Directive

This section contains #line Control Directive topics.

Topics

Name	Description
#line (see page 693)	**Syntax**

3.1.3.6.5.1 #line

Syntax

```
#line integer_constant  <"filename">
```

Description

You can use the **#line** directive to supply line numbers to a program for cross-reference and error reporting. If your program consists of sections derived from some other program file, it is often useful to mark such sections with the line numbers of the original source rather than the normal sequential line numbers derived from the composite program.

The **#line** directive indicates that the following source line originally came from line number integer_constant of filename. Once the filename has been registered, subsequent **#line** commands relating to that file can omit the explicit filename argument.

Macros are expanded in **#line** arguments as they are in the **#include** directive.

The **#line** directive is primarily used by utilities that produce C code as output, and not in human-written code.

3.1.3.6.6 Macros With Parameters Overview

This section contains Macros With Parameters Overview topics.

Topics

Name	Description
Macros With Parameters (see page 694)	The following syntax is used to define a macro with parameters:
Nesting Parentheses And Commas (see page 695)	The actual_arg_list can contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters.
Token Pasting With ## (see page 695)	You can paste (or merge) two tokens together by separating them with ## (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens into one new token. You can use this to construct identifiers. Given the definition
Converting To Strings With # (see page 695)	The # symbol can be placed in front of a formal macro argument in order to convert the actual argument to a string after replacement. Given the following definition:
Using The Backslash (\) For Line Continuation (see page 695)	A long token sequence can straddle a line by using a backslash (\). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions.
Side Effects And Other Dangers (see page 696)	The similarities between function and macro calls often obscure their differences. A macro call has no built-in type checking, so a mismatch between formal and actual argument data types can produce bizarre, hard-to-debug results with no immediate warning. Macro calls can also give rise to unwanted side effects, especially when an actual argument is evaluated more than once. Compare CUBE and cube in the following example.

3.1.3.6.6.1 Macros With Parameters

The following syntax is used to define a macro with parameters:

```
#define macro_identifier(<arg_list>) token_sequence
```

Any comma within parentheses in an argument list is treated as part of the argument, not as an argument delimiter.

Note there can be no whitespace between the macro identifier and the **(**. The optional arg_list is a sequence of identifiers separated by commas, not unlike the argument list of a C function. Each comma-delimited identifier plays the role of a formal argument or placeholder.

Such macros are called by writing

```
macro_identifier<whitespace>(<actual_arg_list>)
```

in the subsequent source code. The syntax is identical to that of a function call; indeed, many standard library C "functions" are implemented as macros. However, there are some important semantic differences, side effects, and potential pitfalls.

The optional actual_arg_list must contain the same number of comma-delimited token sequences, known as actual arguments, as found in the formal arg_list of the **#define** line: There must be an actual argument for each formal argument. An error will be reported if the number of arguments in the two lists is different.

A macro call results in two sets of replacements. First, the macro identifier and the parenthesis-enclosed arguments are replaced by the token sequence. Next, any formal arguments occurring in the token sequence are replaced by the corresponding real arguments appearing in the actual_arg_list.

As with simple macro definitions, rescanning occurs to detect any embedded macro identifiers eligible for expansion.

See Also

Converting To Strings With # (see page 695)

Nesting Parentheses And Commas (see page 695)

Side Effects And Other Dangers (see page 696)

Using The Backslash (\) For Line Continuation (see page 695)

3.1.3.6.6.2 Nesting Parentheses And Commas

The actual_arg_list can contain nested parentheses provided that they are balanced; also, commas appearing within quotes or parentheses are not treated like argument delimiters.

```
#define ERRMSG(x, str) showerr("Error:", x. str)
#define SUM(x,y)  ((x) + (y))
:
ERRMSG(2, "Press Enter, then ESC");
/* showerr("Error",2,"Press Enter, then ESC"); */
return SUM(f(i, j), g(k, l));
/* return ( (f(i, j)) + (g(k, l)) ); */
```

3.1.3.6.6.3 Token Pasting With

You can paste (or merge) two tokens together by separating them with ## (plus optional whitespace on either side). The preprocessor removes the whitespace and the ##, combining the separate tokens into one new token. You can use this to construct identifiers.

Given the definition

```
#define VAR(i, j) (i##j)
```

the call VAR(x, 6) expands to (x6). This replaces the older nonportable method of using (i/**/j).

3.1.3.6.6.4 Converting To Strings With

The # symbol can be placed in front of a formal macro argument in order to convert the actual argument to a string after replacement.

Given the following definition:

```
#define TRACE(flag) printf(#flag "=%d\n", flag)
```

the code fragment

```
int highval = 1024;
TRACE(highval);
```

becomes

```
int highval = 1024;
printf("highval" "=%d\n", highval);
```

which, in turn, is treated as

```
int highval = 1024;
printf("highval=%d\n", highval);
```

3.1.3.6.6.5 Using The Backslash (\) For Line Continuation

A long token sequence can straddle a line by using a backslash (\). The backslash and the following newline are both stripped to provide the actual token sequence used in expansions.

```
#define WARN "This is really a single-\
line warning."
```

3.1.3.6.6.6 Side Effects And Other Dangers

The similarities between function and macro calls often obscure their differences. A macro call has no built-in type checking, so a mismatch between formal and actual argument data types can produce bizarre, hard-to-debug results with no immediate warning. Macro calls can also give rise to unwanted side effects, especially when an actual argument is evaluated more than once.

Compare CUBE and cube in the following example.

```
int cube(int x) {
   return x* x*x;
```

3.1.3.6.7 Pragma Directives Overview

This section contains Pragma Directives Overview topics.

Topics

Name	Description
#pragma (see page 696)	**Syntax**
#pragma alignment (see page 698)	**Syntax**
#pragma anon_struct (see page 698)	**Syntax**
#pragma argsused (see page 699)	**Syntax**
#pragma checkoption (see page 699)	**Syntax**
#pragma codeseg (see page 699)	**Syntax**
#pragma comment (see page 699)	**Syntax**
#pragma defineonoption and #pragma undefineonoption (see page 700)	**Syntax**
#pragma exit and #pragma startup (see page 700)	**Syntax**
#pragma hdrfile (see page 700)	**Syntax**
#pragma hdrstop (see page 701)	**Syntax**
#pragma inline (see page 701)	**Syntax**
#pragma intrinsic (see page 701)	**Syntax**
#pragma link (see page 701)	**Syntax**
#pragma message (see page 702)	**Syntax**
#pragma nopushoptwarn (see page 702)	**Syntax**
#pragma obsolete (see page 702)	**Syntax**
#pragma option (see page 703)	**Syntax**
#pragma pack (see page 704)	**Syntax**
#pragma package (see page 706)	**Syntax**
#pragma region and #pragma end_region (see page 707)	**Syntax**
#pragma resource (see page 707)	**Syntax**
#pragma warn (see page 707)	**Syntax**

3.1.3.6.7.1 #pragma

Syntax

```
#pragma directive-name
```

Description

With **#pragma**, you can set compiler directives in your source code, without interfering with other compilers that also support **#pragma**. If the compiler doesn't recognize directive-name, it ignores the **#pragma** directive without any error or warning message.

The CodeGear C++ supports the following **#pragma** directives: (Links to these topics are at the bottom of this topic.)

- **#pragma alignment**
- **#pragma anon_struct**
- **#pragma argsused**
- **#pragma checkoption**
- **#pragma codeseg**
- **#pragma comment**
- **#pragma defineonoption**
- **#pragma exit**
- **#pragma hdrfile**
- **#pragma hdrstop**
- **#pragma inline**
- **#pragma intrinsic**
- **#pragma link**
- **#pragma message**
- **#pragma nopushoptwarn**
- **#pragma obsolete**
- **#pragma option**
- **#pragma pack**
- **#pragma package**
- **#pragma region/end_region**
- **#pragma resource**
- **#pragma startup**
- **#pragma undefineonoption**
- **#pragma warn**

See Also

#pragma alignment (see page 698)

#pragma anon_struct (see page 698)

#pragma argsused (see page 699)

#pragma checkoption (see page 699)

#pragma codeseg (see page 699)

#pragma comment (see page 699)

#pragma defineonoption (see page 700)

#pragma exit (see page 700)

#pragma hdrfile (see page 700)

#pragma hdrstop (see page 701)

#pragma inline (see page 701)

#pragma intrinsic (see page 701)

#pragma link (see page 701)

#pragma message (see page 702)

#pragma nopushoptwarn (see page 702)

#pragma obsolete (see page 702)

#pragma option (see page 702)

#pragma pack (see page 704)

#pragma package (see page 706)

#pragma region/end_region (see page 707)

#pragma resource (see page 707)

#pragma startup (see page 700)

#pragma undefineonoption (see page 700)

#pragma warn (see page 707)

3.1.3.6.7.2 #pragma alignment

Syntax

```
#pragma alignment
```

Description

This pragma prints out the current byte alignment and `enum` size.

For instance, the pragma

```
#pragma alignment
```

might print out this:

The alignment is 4 bytes, the enum size is 4 bytes

3.1.3.6.7.3 #pragma anon_struct

Syntax

```
#pragma anon_struct on
#pragma anon_struct off
```

Description

The **anon_struct** directive allows you to compile anonymous structures embedded in classes.

```
#pragma anon_struct on
struct S {
   int i;
   struct {  // Embedded anonymous struct
      int   j ;
      float x ;
   };
   class {  // Embedded anonymous class
   public:
      long double ld;
   };
   S() { i = 1; j = 2; x = 3.3; ld = 12345.5;}
};
#pragma anon_struct off
```

```
void main()
{
   S mystruct;
   mystruct.x = 1.2;  // Assign to embedded data.
}
```

3.1.3.6.7.4 #pragma argsused

Syntax

```
#pragma argsused
```

Description

The **argsused** pragma is allowed only between function definitions, and it affects only the next function. It disables the warning message:

```
"Parameter name is never used in function func-name"
```

3.1.3.6.7.5 #pragma checkoption

Syntax

```
#pragma checkoption <switches>
```

where <switches> are any command line switches.

Description

This pragma checks a set of options.

For instance, the following pragma

```
#pragma checkoption -w- -Vx
```

checks if the options `-w-` and `-Vx` are active. If not, you get the error:

Error E2471 **filename**: pragma checkoption failed: options are not as expected

3.1.3.6.7.6 #pragma codeseg

Syntax

```
#pragma codeseg <seg_name> <"seg_class"> <group>
```

Description

The **codeseg** directive lets you name the segment, class, or group where functions are allocated. If the pragma is used without any of its options, the default code segment is used for function allocation.

3.1.3.6.7.7 #pragma comment

Syntax

```
#pragma comment (comment type, "string")
```

Description

The **comment** directive lets you write a comment record into an output file. The comment type can be one of the following values:

Value	Explanation
exestr	The linker writes string into an .obj file. Your specified string is placed in the executable file. Such a string is never loaded into memory but can be found in the executable file by use of a suitable file search utility.

lib	Writes a comment record into an .obj file. A library module that is not specified in the linker's response-file can be specified by the **comment** LIB directive. The linker includes the library module name specified in string as the last library. Multiple modules can be named and linked in the order in which they are named.
user	The compiler writes string into the .obj file. The linker ignores the specified string.

3.1.3.6.7.8 #pragma defineonoption and #pragma undefineonoption

Syntax

```
#pragma defineonoption <macroname>
        <switches>
#pragma undefineonoption <macroname>
        <switches>
```

where <macroname> is a name to define or undefine, and <switches> are any command line switches.

Description

The pragma `#pragma defineonoption` defines a macro name when specified options are set. The macro name is defined if *all* the switches in are turned on, either from the command line or through a pragma.

The pragma `#pragma undefineonoption` undefines a macro name when specified options are set.

For instance, the following pragma

```
#pragma defineonoption DEBUG -v
```

results in the macro `DEBUG` being defined when the option `-v` is set.

In contrast, this pragma

```
#pragma undefineonoption DEBUG -v
```

results in the macro `DEBUG` being *undefined* when the option `-v` is set.

3.1.3.6.7.9 #pragma exit and #pragma startup

Syntax

```
#pragma startup function-name <priority>
#pragma exit function-name <priority>
```

Description

These two pragmas allow the program to specify function(s) that should be called either upon program startup (before the main function is called), or program exit (just before the program terminates through _**exit**).

The specified function-name must be a previously declared function taking no arguments and returning **void**; in other words, it should be declared as:

```
void func(void);
```

The optional priority parameter should be an integer in the range 64 to 255. The highest priority is 0. Functions with higher priorities are called first at startup and last at exit. If you don't specify a priority, it defaults to 100.

Warning: Do not use priority values less than 64. Priorities from 0 to 63 are reserved for ISO startup and shutdown mechanisms.

3.1.3.6.7.10 #pragma hdrfile

Syntax

```
#pragma hdrfile "filename.csm"
```

Description

This directive sets the name of the file in which to store precompiled headers.

If you aren't using precompiled headers, this directive has no effect. You can use the command-line compiler option -H=filename or Use Precompiled Headers to change the name of the file used to store precompiled headers.

3.1.3.6.7.11 #pragma hdrstop

Syntax

```
#pragma hdrstop
```

Description

This directive terminates the list of header files eligible for precompilation. You can use it to reduce the amount of disk space used by precompiled headers.

Precompiled header files can be shared between the source files of your project only if the **#include** directives before **#pragma hdrstop** are identical. Therefore, you get the best compiler performance if you include common header files of your project before the **#pragma hdrstop**, and specific ones after it. Make sure the **#include** directives before the **#pragma hdrstop** are identical in all the source files, or that there are only very few variations.

The integrated development environment generates code to enhance precompiled header performance. For example, after a New Application, source file "Unit1.cpp" will look like this (comments added):

```
#include <clxvcl.h>       // common header file
#pragma hdrstop      // terminate list here
```

Use this pragma directive only in source files. The pragma has no effect when it is used in a header file.

3.1.3.6.7.12 #pragma inline

Syntax

```
#pragma inline
```

Description

This directive is equivalent to the -B command-line compiler option.

This is best placed at the top of the file, because the compiler restarts itself with the -B option when it encounters **#pragma inline**.

3.1.3.6.7.13 #pragma intrinsic

Syntax

```
#pragma intrinsic [-]function-name
```

Description

Use **#pragma intrinsic** to override command-line switches or project options to control the inlining of functions.

When inlining an intrinsic function, always include a prototype for that function before using it.

3.1.3.6.7.14 #pragma link

Syntax

```
#pragma link "[path]modulename[.ext]"
```

Description

The directive instructs the linker to link the file into an executable file.

By default, the linker searches for modulename in the local directory and any path specified by the **-L** option. You can use the path argument to specify a directory.

By default, the linker assumes a .obj extension.

3.1.3.6.7.15 #pragma message

Syntax

```
#pragma message ("text" ["text"["text" ...]])
#pragma message text
```

Description

Use **#pragma message** to specify a user-defined message within your program code.

The first form requires that the text consist of one or more string constants, and the message must be enclosed in parentheses. (This form is compatible with Microsoft C.) This form will output the constant contained between the double quotes regardless of whether it is a macro or not.

The second form uses the text following the **#pragma** for the text of the warning message. With this form of the **#pragma**, any macro references are expanded before the message is displayed.

The third form will output the macro-expanded value of text following the **#pragma**, if it is **#defined**. If is not **#defined**, you'll get an ill-formed pragma warning.

User-defined messages are displayed as messages, not warnings.

Display of user-defined messages is on by default and can be turned on or off with the `Show Messages` option. This option corresponds to the compiler's **-wmsg** switch.

Messages are only displayed in the IDE if **Show general messages** is checked on the **C++ Project Properties** under **Project ▸ Options ▸ Project Propertes**.

See Also

Preprocessor Directives (see page 710)

3.1.3.6.7.16 #pragma nopushoptwarn

Syntax

```
#pragma nopushoptwarn
```

Description

The **nopushoptwarn** pragma allows the use of pragma option push without pop and pragma option pop without push // in a file. Normally a warning would be given.

3.1.3.6.7.17 #pragma obsolete

Syntax

```
#pragma obsolete identifier
```

Description

Use **#pragma obsolete** to give a warning the first time the preprocessor encounters identifier in your program code after the

pragma. The warning states that the identifier is obsolete.

3.1.3.6.7.18 #pragma option

Syntax

```
#pragma option options
#pragma option push options
#pragma option pop
#pragma nopushoptwarn
```

Description

Use **#pragma option** to include command-line options within your program code. **#pragma option** can also be used with the push or pop arguments.

#pragma option [options...]

options can be any command-line option (except those listed in the following paragraph). Any number of options can appear in one directive. For example, both of the following are valid:

```
#pragma option -C
#pragma option -C -A
```

Any of the toggle options (such as **-a** or **-K**) can be turned on and off as on the command line. For these toggle options, you can also put a period following the option to return the option to its command-line, configuration file, or option-menu setting. This allows you to temporarily change an option, and then return it to its default, without having to remember (or even needing to know) what the exact default setting was.

Options that cannot appear in a **pragma option** include:

B	c	dname
Dname=string	efilename	E
Fx	h	Ifilename
lexset	M	o
P	Q	S
T	Uname	V
X	Y	

- You can use **#pragma**s, **#include**s, **#define**, and some **#if**s in the following cases:
- Before the use of any macro name that begins with two underscores (and is therefore a possible built-in macro) in an **#if**, **#ifdef**, **#ifndef** or **#elif** directive.
- Before the occurrence of the first real token (the first C or C++ declaration).

Certain command-line options can appear only in a #**pragma option** command before these events. These options are:

Efilename	f	i#
m*	npath	ofilename
U	W	z

*

Other options can be changed anywhere. The following options will only affect the compiler if they get changed between functions or object declarations:

1	h	r
2	k	rd
A	N	v
Ff	O	y
G	p	Z

The following options can be changed at any time and take effect immediately:

A	gn	zE
b	jn	zF
C	K	zH
d	wxxx	

The options can appear followed by a dot (.) to reset the option to its command-line state.

#pragma option using push or pop

The **#pragma option** directive can also be used with the push and pop arguments to enable you to easily modify compiler directives.

Using the **#pragma option push** allows you to save all (or a selected subset of) options before including any files that could potentially change many compiler options and warnings, and then, with the single statement, **#pragma option pop**, return to the previous state. For example:

```
#pragma option push
#include <theworld.h>
#pragma option pop
#include "mystuff.h"
```

The **#pragma option push** directive first pushes all compiler options and warning settings on a stack and then handles any options (if supplied). The following examples show how the **#pragma option push** can be used with or without options:

```
#pragma option push -C -A
#pragma option push
```

The **#pragma option pop** directive changes compiler options and warnings by popping the last set of options and warnings from the stack. It gives a warning, "Pragma option pop with no matching option push", if the stack is empty, in which case nothing happens.

The following generates a warning about an empty stack:

```
#pragma option push
#pragma option pop
#pragma option pop      /* Warning */
```

We do not recommend it, but you can turn off this warning with the directive: #pragma warn -nop.

If you try to specify any options after pop, you get the error, "Nothing allowed after pragma option pop." For example, the following produces an error:

```
#pragma option pop -C/* ERROR */
```

If your stack of pushed options is not the same at the start of a file as at the end of a file, you receive a warning: "Previous options and warnings not restored." To turn off this warning, use the directive, **#pragma nopushoptwarn** .

3.1.3.6.7.19 #pragma pack

Syntax

```
#pragma pack([{push | pop}[,]] [identifier[,]] [n])
```

Description

The **#pragma pack(n)**.directive is the same as using the #pragma option specifically with the -a compiler option. **n** is the byte alignment that determines how the compiler aligns data in stored memory. For more information see the -a compiler option. **#pragma pack** can also be used with **push** and **pop** arguments, which provide the same functionality as the **#pragma option** directive using **push** and **pop**. The following table compares the use of **#pragma pack** with **#pragma option**.

#pragma pack	#pragma option
#pragma pack(n)	#pragma option -an
#pragma pack(push, n)	#pragma option push -an
#pragma pack(pop)	#pragma option pop

The **#pragma pack** directive also supports an identifier argument which must be used in combination with either push or pop.

#pragma pack with no arguments

```
#pragma pack()
```

Using **#pragma pack** with no arguments will set the packing size to the starting –aX alignment (which defaults to 8). The starting -aX alignment is considered the alignment at the start of the compile AFTER all command-line options have been processed.

#pragma pack using a value for n

```
#pragma pack(8)
```

Using **#pragma pack** with a value for 'n' will set the current alignment to 'n'. Valid alignments for ‘n’ are: 1,2,4,8, and 16.

#pragma pack using push

```
#pragma pack(push)
```

Using **#pragma pack** with push will push the current alignment on an internal stack of alignments.

#pragma pack using push, identifier

```
#pragma pack(push, ident)
```

Using **#pragma pack** with push and an identifier will associate the pushed alignment with 'identifier'.

#pragma pack using push and n

```
#pragma pack(push, 8)
#pragma pack(push, ident, 8)
```

Using **#pragma pack** with push with a value for 'n', will execute pragma pack push or pragma pack push identifier, and afterwards set the current alignment to 'n'.

#pragma pack using pop

```
#pragma pack(pop)
```

Using **#pragma pack** with pop will pop the alignment stack and set the alignment to the last alignment pushed. If the pop does not find a corresponding push, the entire stack of alignments is popped, and a warning is issued, and the alignment reverts to the starting -aX alignment..

#pragma pack using pop, identifier

```
#pragma pop(pop, ident)
```

Using **#pragma pack** with pop and an identifier will pop the stack until the identifier is found and set the alignment to the alignment pushed by the previous corresponding **#pragma pack(push, identifier)**. If the pop with identifier does not find the corresponding push with an identifier, the entire stack of alignments is popped, and a warning is issued to the user:

```
W8083: Pragma pack pop with no matching pack push
```

The alignment will then be reset to the starting -aX alignment.

#pragma pack using pop and n

```
#pragma pack(pop, 8)
#pragma pack(pop, ident, 8)
```

Using **#pragma pack** with pop and a value for 'n', will execute pragma pack pop or pragma pack pop identifier. Afterwards the current alignment is set to 'n', unless pop fails to find a corresponding push, in which case 'n' is ignored, a warning is issued, and the alignment reverts to the starting -aX alignment.

Error conditions

Specifying an 'identifier' without push or pop is an error.

Specifying an alignment different from 1,2,4,8,16 is an error.

Warning conditions

Using **#pragma pop** without a corresponding push issues a warning.

3.1.3.6.7.20 #pragma package

Syntax

```
#pragma package(smart_init)
#pragma package(smart_init, weak)
```

Description: smart_init argument

The **#pragma package(smart_init)** assures that packaged units are initialized in the order determined by their dependencies. (Included by default in package source file.) Typically, you would use the **#pragma package** for .cpp files that are built as packages.

This pragma affects the order of initialization of that compilation unit. For units, initialization occurs in the following order:

- 1. By their "uses" dependencies, that is, if unitA depends on unitB, unitB must be initialized before unitA.
- 2. The link order.
- 3. Priority order within the unit.

For regular object files (those not built as units), initialization happens first according to priority order and then link order. Changing the link order of the object files changes the order in which the global object constructors get called.

The following examples show how the initialization differs between units and regular object files.

Take as an example three unit files, A, B and C that are "smart initialized" with **#pragma package(smart_init)** and have priority values (defined by the priority parameter of the #pragma startup) set of 10, 20 and 30. The functions are named according to their priority value and the parent object file, so the names are a10, a20, a30, b10, and so on.

Since all three are units, and if A uses B and C and the link order is A, B then C, the order of initialization is:

B10 B20 B30 C10 C20 C30 A10 A20 A30

If the above were object files, not units, the order would be:

A10 B10 C10 A20 B20 C20 A30 B30 C30

The .cpp files that use **#pragma package(smart_init)** also require that any **#pragma link** references to other object files from a .cpp file that declares **#pragma package(smart_init)**, must be resolved by a unit. **#pragma link** references to non object files can still be resolved by libraries, etc.

Description: weak packages

The **#pragma package(smart_init, weak)** directive affects the way an object file is stored in a package's .bpi and .bpl files. If **#pragma package(smart_init, weak)** appears in a unit file, the compiler omits the unit from BPLs when possible, and creates

a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be "weakly packaged".

#pragma package(smart_init, weak) is used to eliminate conflicts among packages that may depend on the same external library.

Unit files containing the **#pragma package(smart_init, weak)** directive must not have global variables.

For more information about using weak packages, see Weak packaging.

See Also

Preprocessor Directives (see page 710)

3.1.3.6.7.21 #pragma region and #pragma end_region

Syntax

```
#pragma region <name> #pragma end_region
```

Description

These two pragmas allow the program to specify a region that can be collapsed and expanded, similar to the facility automatically provided for functions.

Use these pragmas by enclosing the code section as follows:

```
#pragma region name code to be collapsed/expanded ... #pragma
        end_region
```

The name is optional.

3.1.3.6.7.22 #pragma resource

Syntax

```
#pragma resource "*.xdfm"
```

Description

This pragma causes the file to be marked as a form unit and requires matching .dfm and header files. All such files are managed by the IDE.

If your form requires any variables, they must be declared immediately after the pragma resource is used. The declarations must be of the form

TFormName *Formname;

3.1.3.6.7.23 #pragma warn

Syntax

```
#pragma warn [+|-|.]xxx
```

Description

The **warn** pragma lets you override specific **-wxxx** command-line options or check Display Warnings. **xxx** is the three-letter or four-digit message identifier used by the command-line option.

+ turns on a warning while - disables the warning. The '.' restores a warning to the setting in Project Options.

3.1.3.6.8 Predefined Macros Overview

This section contains Predefined Macros Overview topics.

Topics

Name	Description
Predefined Macros (see page 708)	The compiler predefines certain global identifiers, known as manifest constants. Most global identifiers begin and end with two underscores (__). **Note:** For readability, underscores are often separated by a single blank space. In your source code, you should never insert whitespace between underscores.

3.1.3.6.8.1 Predefined Macros

The compiler predefines certain global identifiers, known as manifest constants. Most global identifiers begin and end with two underscores (__).

Note: For readability, underscores are often separated by a single blank space. In your source code, you should never insert whitespace between underscores.

Macro	Value	Description
__BCOPT__	1	Defined only in compilers that support optimization, therefore always defined.
__BCPLUSPLUS__	• 0x0570 for BDS 2006 • 0x0590 for C++Builder 2007 • 0x0591 for update 1 to C++Builder 2007 • 0x0592 for RAD Studio 2007 • 0x0593 for the December update to RAD Studio 2007 • 0x0610 for C++Builder 2009 and for C++Builder 2009 Update 1	Defined if you've selected C++ compilation; will increase in later releases.
__BOOL__	1	Indicates that the **bool** keyword is accepted.
__BORLANDC__	• 0x0570 for BDS 2006 • 0x0590 for C++Builder 2007 • 0x0591 for update 1 to C++Builder 2007 • 0x0592 for RAD Studio 2007 • 0x0593 for the December update to RAD Studio 2007 • 0x0610 for C++Builder 2009 and for C++Builder 2009 Update 1	Version number.
__CDECL__	1	Defined if Calling Convention is set to cdecl; otherwise undefined.

_CHAR_UNSIGNED	1	Undefined by default. Using the -K switch to make the default character unsigned causes this macro to be defined. Ultimately, the -K option controls how a **char** is extended when converted to an **int**. By default, the compiler sign-extends, but if you enable _CHAR_UNSIGNED_, then the compiler zero-extends characters when converting to **int**.
__CODEGEARC__	0x0610 for C++Builder 2009	Version number.
__CODEGUARD__		Defined whenever one of the CodeGuard compiler options is used; otherwise it is undefined.
__CONSOLE__	1	When defined, the macro indicates that the program is a console application.
_CPPUNWIND	1	Enable stack unwinding. This is true by default; use **-xd-**!ALink(OSCGExceptions1) to disable.
__cplusplus	1	Defined if in C++ mode; otherwise, undefined.
__DATE__	String literal	Date when processing began on the current file.
__DLL__	1	Defined whenever the -WD compiler option is used; otherwise it is undefined.
__FILE__	String literal	Name of the current file being processed.
__FLAT__	1	Defined when compiling in 32-bit flat memory model.
__FUNC__	String literal	Name of the current function being processed. More details.
__LINE__	Decimal constant	Number of the current source file line being processed.
_M_IX86	0x12c	Always defined. The default value is 300. You can change the value to 400 or 500 by using the /4 or /5 compiler options.
__MT__	1	Defined only if the -tWM option is used. It specifies that the multithread library is to be linked.
__PASCAL__	1	Defined if Calling Convention is set to Pascal; otherwise undefined.
_PUSHPOP_SUPPORTED	1	Always defined; allows Microsoft standard headers to use push and pop to verify whether a feature is supported.
_STDCALL_SUPPORTED	1	Always defined; defines the Microsoft stdcall calling convention.
__STDC__	1	Defined if you compile with the -A compiler option; otherwise, it is undefined.
__TCPLUSPLUS__	0x0570 0x0610 for C++Builder 2009	Version number.
__TEMPLATES__	1	Defined as 1 for C++ files (meaning that templates are supported); otherwise, it is undefined.
__TIME__	String literal	Time when processing began on the current file.
__TLS__	1	Thread Local Storage. Always true.

__TURBOC__	• 0x0570 for BDS 2006 • 0x0590 for C++Builder 2007 • 0x0591 for update 1 to C++Builder 2007 • 0x0592 for RAD Studio 2007 • 0x0593 for the December update to RAD Studio 2007 • 0x0610 for C++Builder 2009 and for C++Builder 2009 Update 1	Will increase in later releases.
_WCHAR_T		Defined only for C++ programs to indicate that **wchar_t** is an intrinsically defined data type.
_WCHAR_T_DEFINED		Defined only for C++ programs to indicate that **wchar_t** is an intrinsically defined data type.
_Windows	1	Defined when compiling on the Windows platform.
__WIN32__	1	Defined for console and GUI applications on the Windows platform.

Note: The predefined macros __DATE__

, **__FILE__** , **__FUNC__**, **__LINE__**, **__STDC__**, and **__TIME__** cannot be redefined or undefined.

See Also

Side Effects And Other Dangers (see page 696)

Using The Backslash (\) For Line Continuation (see page 695)

Converting To Strings With # (see page 695)

3.1.3.6.9 Preprocessor Directives

This section contains C++ Preprocessor Directive topics.

Topics

Name	Description
Preprocessor Directives (see page 710)	Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program. The preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them. The preprocessor supports these directives:
# (null Directive) (see page 711)	**Syntax**

3.1.3.6.9.1 Preprocessor Directives

Preprocessor directives are usually placed at the beginning of your source code, but they can legally appear at any point in a program. The preprocessor detects preprocessor directives (also known as control lines) and parses the tokens embedded in them. The preprocessor supports these directives:

# (null directive)	#ifdef (see page 688)
#define (see page 690)	#ifndef (see page 688)
#elif (see page 688)	#undef (see page 691)
#else (see page 688)	#include (see page 692)
#endif (see page 688)	#line (see page 693)
#error (see page 692)	#pragma (see page 696)
#if (see page 688)	#import (see page 692)

Any line with a leading # is taken as a preprocessing directive, unless the # is within a string literal, in a character constant, or embedded in a comment. The initial # can be preceded or followed by whitespace (excluding new lines).

3.1.3.6.9.2 # (null Directive)

Syntax

#

Description

The null directive consists of a line containing the single character #. This line is always ignored.

3.1.4 C Runtime Library Reference

RAD Studio has several hundred functions, macros, and classes that you call from within your C and C++ programs to perform a wide variety of tasks, including low- and high-level I/O, string and file manipulation, memory allocation, process control, data conversion, mathematical calculations, and more.

Note: In the online help, each function, macro, and class in the C Runtime Library is listed only once . However, some functions, macros, and classes are defined in more than one header file.

For example, `_strerror` is defined in both string.h and stdio.h. For functions that are defined in several header files, the online help assigns the function to the first header file in alphabetic order. Thus, you will find `_strerror` listed in the help in stdio.h but not in string.h.

To find a function, macro, or class that is not listed in a header file in which you expect it, please use the help Index.

Topics

Name	Description
alloc.h (see page 712)	The following functions, macros, and classes are provided in `alloc.h`:
assert.h (see page 730)	The following functions, macros, and classes are provided in `assert.h`:
conio.h (see page 732)	The following functions, macros, and classes are provided in `conio.h`:
ctype.h (see page 766)	The following functions, macros, and classes are provided in `ctype.h`:
delayimp.h (see page 787)	The following functions, macros, and classes are provided in `delayimp.h`:
direct.h (see page 790)	The following functions, macros, and classes are provided in `direct.h`:
dirent.h (see page 792)	The following functions, macros, and classes are provided in `dirent.h`:
dir.h (see page 799)	The following functions, macros, and classes are provided in `dir.h`:
dos.h (see page 818)	The following functions, macros, and classes are provided in `dos.h`:
errno.h (see page 831)	The following functions, macros, and classes are provided in `errno.h`:
except.h (see page 838)	The following functions, macros, and classes are provided in `except.h`:
fastmath.h (see page 842)	The following functions, macros, and classes are provided in `fastmath.h`:

fcntl.h (see page 844)	The following functions, macros, and classes are provided in `fcntl.h`:
float.h (see page 853)	The following functions, macros, and classes are provided in `float.h`:
io.h (see page 864)	The following functions, macros, and classes are provided in `io.h`:
limits.h (see page 913)	The following functions, macros, and classes are provided in `limits.h`:
locale.h (see page 915)	The following functions, macros, and classes are provided in `locale.h`:
malloc.h (see page 920)	The following functions, macros, and classes are provided in `malloc.h`:
math.h (see page 921)	The following functions, macros, and classes are provided in `math.h`:
mem.h (see page 959)	The following functions, macros, and classes are provided in `mem.h`:
new.h (see page 968)	The following functions, macros, and classes are provided in `new.h`:
process.h (see page 971)	The following functions, macros, and classes are provided in `process.h`:
setjmp.h (see page 993)	The following functions, macros, and classes are provided in `setjmp.h`:
share.h (see page 996)	The following functions, macros, and classes are provided in `share.h`:
signal.h (see page 997)	The following functions, macros, and classes are provided in `signal.h`:
stdarg.h (see page 1003)	The following functions, macros, and classes are provided in `stdarg.h`:
stddef.h (see page 1004)	The following functions, macros, and classes are provided in `stddef.h`:
stdio.h (see page 1006)	The following functions, macros, and classes are provided in `stdio.h`:
stdlib.h (see page 1080)	The following functions, macros, and classes are provided in `stdlib.h`:
string.h (see page 1139)	The following functions, macros, and classes are provided in `string.h`:
sys\stat.h (see page 1186)	The following functions, macros, and classes are provided in `sys\stat.h`:
sys\timeb.h (see page 1190)	The following functions, macros, and classes are provided in `sys\timeb.h`:
sys\types.h (see page 1192)	The following functions, macros, and classes are provided in `sys\types.h`:
time.h (see page 1193)	The following functions, macros, and classes are provided in `time.h`:
typeinfo.h (see page 1211)	The following functions, macros, and classes are provided in `typeinfo.h`:
utime.h (see page 1213)	The following functions, macros, and classes are provided in `utime.h`:
values.h (see page 1214)	The following functions, macros, and classes are provided in `values.h`:

3.1.4.1 alloc.h

The following functions, macros, and classes are provided in `alloc.h`:

Topics

Name	Description
_heapchk (see page 716)	**Header File** malloc.h **Category** Memory Routines **Syntax** `int _heapchk(void);` **Description** Checks and verifies the heap. _heapchk walks through the heap and examines each block, checking its pointers, size, and other critical attributes. **Return Value** One of the following values:

_heapmin (see page 717)	**Header File** malloc.h **Category** Memory Routines **Prototype** `int _heapmin(void);` **Description** Release unused heap areas. The _heapmin function returns unused areas of the heap to the operating system. This allows other processes to use blocks that have been allocated and then freed. Due to fragmentation of the heap, _heapmin might not always be able to return unused memory to the operating system; this is not an error. **Return Value** _heapmin returns 0 if it is successful, or -1 if an error occurs. **Portability**
_heapset (see page 717)	**Header File** malloc.h **Category** Memory Routines **Prototype** `int _heapset(unsigned int fillvalue);` **Description** Fills the free blocks on the heap with a constant value. _heapset checks the heap for consistency using the same methods as _heapchk. It then fills each free block in the heap with the value contained in the least significant byte of fillvalue. This function can be used to find heap-related problems. It does not guarantee that subsequently allocated blocks will be filled with the specified value. **Return Value** One of the following values:
_msize (see page 720)	**Header File** malloc.h **Category** Memory Routines **Prototype** `size_t _msize(void *block);` **Description** Returns the size of a heap block. _msize returns the size of the allocated heap block whose address is block. The block must have been allocated with malloc, calloc, or realloc. The returned size can be larger than the number of bytes originally requested when the block was allocated. **Return Value** _msize returns the size of the block in bytes. **Example**
_rtl_heapwalk (see page 720)	**Header File** malloc.h **Category** Memory Routines **Prototype** `int _rtl_heapwalk(_HEAPINFO *hi);` **Description** Inspects the heap node by node. **Note:** This function replaces _heapwalk which is obsolete. _rtl_heapwalk assumes the heap is correct. Use _heapchk to verify the heap before using _rtl_heapwalk. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to _rtl_heapwalk. _rtl_heapwalk receives a pointer to a structure of type _HEAPINFO (declared in malloc.h). Note that the _HEAPINFO structure must be allocated on the heap (using malloc()). You can't pass the address of a variable declared on the stack. For the... more (see page 720)

calloc (see page 721)	**Header File** alloc.h, stdlib.h **Category** Memory Routines **Prototype** `void *calloc(size_t nitems, size_t size);` **Description** Allocates main memory. calloc provides access to the C memory heap. The heap is available for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ heap memory allocation. calloc allocates a block of size nitems * size. The block is initialized to 0. **Return Value** calloc returns a pointer to the newly allocated block. If not enough space exists for the new block or if nitems or size is 0, calloc returns NULL. **Example**
free (see page 722)	**Header File** alloc.h, stdlib.h **Category** Memory Routines **Prototype** `void free(void *block);` **Description** Frees allocated block. free deallocates a memory block allocated by a previous call to calloc, malloc, or realloc. **Return Value** None. **Example**
heapcheck (see page 723)	**Header File** alloc.h **Category** Memory Routines **Prototype** `int heapcheck(void);` **Description** Checks and verifies the heap. heapcheck walks through the heap and examines each block, checking its pointers, size, and other critical attributes. **Return Value** The return value is less than 0 for an error and greater than 0 for success. The return values and their meaning are as follows:
heapcheckfree (see page 724)	**Header File** alloc.h **Category** Memory Routines **Prototype** `int heapcheckfree(unsigned int fillvalue);` **Description** Checks the free blocks on the heap for a constant value. **Return Value** The return value is less then 0 for an error and greater than 0 for success. The return values and their meaning are as follows:

heapchecknode (see page 725)	**Header File** alloc.h **Category** Memory Routines **Prototype** `int heapchecknode(void *node);` **Description** Checks and verifies a single node on the heap. If a node has been freed and heapchecknode is called with a pointer to the freed block, heapchecknode can return _BADNODE rather than the expected _FREEENTRY. This is because adjacent free blocks on the heap are merged, and the block in question no longer exists. **Return Value** One of the following values:
heapfillfree (see page 726)	**Header File** alloc.h **Category** Memory Routines **Prototype** `int heapfillfree(unsigned int fillvalue);` **Description** Fills the free blocks on the heap with a constant value. **Return Value** One of the following values:
heapwalk (see page 727)	**Header File** alloc.h **Category** Memory Routines **Prototype** `int heapwalk(struct heapinfo *hi);` **Description** heapwalk is used to “walk” through the heap, node by node. heapwalk assumes the heap is correct. Use heapcheck to verify the heap before using heapwalk. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to heapwalk. heapwalk receives a pointer to a structure of type heapinfo (declared in alloc.h). For the first call to heapwalk, set the hi.ptr field to null. heapwalk returns with hi.ptr containing the address of the first block. hi.size holds the size of the... more (see page 727)
malloc (see page 728)	**Header File** alloc.h, stdlib.h **Category** Memory Routines **Prototype** `void *malloc(size_t size);` **Description** malloc allocates a block of size bytes from the memory heap. It allows a program to allocate memory explicitly as it’s needed, and in the exact amounts needed. Allocates main memory.The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures, for example, trees and lists, naturally employ heap memory allocation. In the large data models, all the space beyond the program stack to the end of available memory is available for the heap. **Return Value** On success, malloc returns a pointer to the... more (see page 728)

realloc (see page 729)	**Header File** alloc.h, stdlib.h **Category** Memory Routines **Prototype** `void *realloc(void *block, size_t size);` **Description** Reallocates main memory. realloc attempts to shrink or expand the previously allocated block to size bytes. If size is zero, the memory block is freed and NULL is returned. The block argument points to a memory block previously obtained by calling malloc, calloc, or realloc. If block is a NULL pointer, realloc works just like malloc. realloc adjusts the size of the allocated block to size, copying the contents to a new location if necessary. **Return Value** realloc returns the address of the reallocated block, which... more (see page 729)
stackavail (see page 729)	**Header File** malloc.h **Category** Memory Routines **Prototype** `size_t stackavail(void);` **Description** Gets the amount of available stack memory. stackavail returns the number of bytes available on the stack. This is the amount of dynamic memory that alloca can access. **Return Value** stackavail returns a size_t value indicating the number of bytes available. **Example**

3.1.4.1.1 _heapchk

Header File

malloc.h

Category

Memory Routines

Syntax

```
int _heapchk(void);
```

Description

Checks and verifies the heap.

_heapchk walks through the heap and examines each block, checking its pointers, size, and other critical attributes.

Return Value

One of the following values:

_HEAPBADNODE	A corrupted heap block has been found
_HEAPEMPTY	No heap exists
_HEAPOK	The heap appears to be uncorrupted

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.1.2 _heapmin

Header File

malloc.h

Category

Memory Routines

Prototype

```
int _heapmin(void);
```

Description

Release unused heap areas.

The _heapmin function returns unused areas of the heap to the operating system. This allows other processes to use blocks that have been allocated and then freed. Due to fragmentation of the heap, _heapmin might not always be able to return unused memory to the operating system; this is not an error.

Return Value

_heapmin returns 0 if it is successful, or -1 if an error occurs.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.1.3 _heapset

Header File

malloc.h

Category

Memory Routines

Prototype

```
int _heapset(unsigned int fillvalue);
```

Description

Fills the free blocks on the heap with a constant value.

_heapset checks the heap for consistency using the same methods as _heapchk. It then fills each free block in the heap with the value contained in the least significant byte of fillvalue. This function can be used to find heap-related problems. It does not guarantee that subsequently allocated blocks will be filled with the specified value.

Return Value

One of the following values:

_HEAPOK	The heap appears to be uncorrupted
_HEAPEMPTY	No heap exists
_HEAPBADNODE	A corrupted heap block has been found

Example

```
#include <windowsx.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
BOOL InitApplication(HINSTANCE hInstance);
HWND InitInstance(HINSTANCE hInstance, int nCmdShow);
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam);
void ExampleHeapSet(HWND hWnd);
#pragma argsused
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
MSG msg;      // message
  if (!InitApplication(hInstance))  // Initialize shared things
   return (FALSE); // Exits if unable to initialize
  /* Perform initializations that apply to a specific instance */
  if (!(InitInstance(hInstance, nCmdShow)))
   return (FALSE);
  /* Acquire and dispatch messages until a WM_QUIT message is received. */
  while (GetMessage(&msg, // message structure
   NULL, // handle of window receiving the message
   NULL, // lowest message to examine
   NULL))  // highest message to examine
  {
  TranslateMessage(&msg); // Translates virtual key codes
  DispatchMessage(&msg);  // Dispatches message to window
  }
  return (msg.wParam);  // Returns the value from PostQuitMessage
}
BOOL InitApplication(HINSTANCE hInstance)
{
  WNDCLASS  wc;
  // Fill in window class structure with parameters that describe the
  // main window.
  wc.style = CS_HREDRAW | CS_VREDRAW; // Class style(s).
  wc.lpfnWndProc = (long (FAR PASCAL*)(void *,unsigned int,unsigned int, long ))MainWndProc;
// Function to retrieve messages for
                // windows of this class.
  wc.cbClsExtra = 0;  // No per-class extra data.
  wc.cbWndExtra = 0;  // No per-window extra data.
  wc.hInstance = hInstance; // Application that owns the class.
  wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName = NULL;  // Name of menu resource in .RC file.
  wc.lpszClassName = "Example"; // Name used in call to CreateWindow.
  /* Register the window class and return success/failure code. */
  return (RegisterClass(&wc));
}
HWND InitInstance(HINSTANCE hInstance, int nCmdShow)
{
  HWND  hWnd; // Main window handle.
  /* Create a main window for this application instance.  */
  hWnd = CreateWindow(
   "Example",                 // See RegisterClass() call.
   "Example _heapset 32 bit only",  // Text for window title bar.
```

```
    WS_OVERLAPPEDWINDOW,        // Window style.
    CW_USEDEFAULT,          // Default horizontal position.
    CW_USEDEFAULT,           // Default vertical position.
    CW_USEDEFAULT,             // Default width.
    CW_USEDEFAULT,         // Default height.
    NULL,                   // Overlapped windows have no parent.
    NULL,                   // Use the window class menu.
    hInstance,               // This instance owns this window.
    NULL                    // Pointer not needed.
   );
   /* If window could not be created, return "failure" */
   if (!hWnd)
    return (FALSE);
   /* Make the window visible; update its client area; and return "success" */
   ShowWindow(hWnd, nCmdShow); // Show the window
   UpdateWindow(hWnd);     // Sends WM_PAINT message
   return (hWnd);        // Returns the value from PostQuitMessage
}
void ExampleHeapSet(HWND hWnd)
{
  int hsts;
  char *buffer;
  if ( (buffer = (char *)malloc( 1 )) == NULL )
    exit(0);
  hsts = _heapset( 'Z' );
  switch (hsts)
  {
    case _HEAPOK:
      MessageBox(hWnd,"Heap is OK","Heap",MB_OK|MB_ICONINFORMATION);
      break;
    case _HEAPEMPTY:
      MessageBox(hWnd,"Heap is empty","Heap",MB_OK|MB_ICONINFORMATION);
      break;
    case _HEAPBADNODE:
      MessageBox(hWnd,"Bad node in heap","Heap",MB_OK|MB_ICONINFORMATION);
      break;
    default:
      break;
  }
  free (buffer);
}
#pragma argsused
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam)
{
  switch (message) {
    case WM_CREATE:
    {
      //Example _heapset
      ExampleHeapSet(hWnd);
      return NULL;
    }
    case WM_QUIT:
    case WM_DESTROY:  // message: window being destroyed
      PostQuitMessage(0);
    break;
    default:      // Passes it on if unprocessed
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.1.4 _msize

Header File

malloc.h

Category

Memory Routines

Prototype

```
size_t _msize(void *block);
```

Description

Returns the size of a heap block.

_msize returns the size of the allocated heap block whose address is block. The block must have been allocated with malloc, calloc, or realloc. The returned size can be larger than the number of bytes originally requested when the block was allocated.

Return Value

_msize returns the size of the block in bytes.

Example

```
#include <malloc.h>        /* malloc() _msize() */
#include <stdio.h>            /* printf() */
int main( )
{
  int size;
  int *buffer;
  buffer = malloc(100 * sizeof(int));
  size = _msize(buffer);
  printf("Allocated %d bytes for 100 integers\n", size);
  return(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.1.5 _rtl_heapwalk

Header File

malloc.h

Category

Memory Routines

Prototype

```
int _rtl_heapwalk(_HEAPINFO *hi);
```

Description

Inspects the heap node by node.

Note: This function replaces _heapwalk which is obsolete.

_rtl_heapwalk assumes the heap is correct. Use _heapchk to verify the heap before using _rtl_heapwalk. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to _rtl_heapwalk.

_rtl_heapwalk receives a pointer to a structure of type _HEAPINFO (declared in malloc.h). Note that the _HEAPINFO structure must be allocated on the heap (using malloc()). You can't pass the address of a variable declared on the stack.

For the first call to _rtl_heapwalk, set the hi._pentry field to NULL. _rtl_heapwalk returns with hi._pentry containing the address of the first block.

hi._size	holds the size of the block in bytes.
hi._useflag	is a flag that is set to _USEDENTRY if the block is currently in use. If the block is free, hi._useflag is set to _FREEENTRY.

Return Value

This function returns one of the following values:

_HEAPBADNODE	A corrupted heap block has been found
_HEAPBADPTR	The _pentry field does not point to a valid heap block
_HEAPEMPTY	No heap exists
_HEAPEND	The end of the heap has been reached
_HEAPOK	The _heapinfo block contains valid information about the next heap block

Example

```
#include <stdio.h>
#include <malloc.h>
#include <alloc.h>
#define NUM_PTRS  10
#define NUM_BYTES 16
int main( void )
{
   _HEAPINFO *hi;
   char *array[ NUM_PTRS ];
   int i;
   hi = (_HEAPINFO *) malloc( sizeof(_HEAPINFO) );
   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );
   for( i = 0; i < NUM_PTRS; i += 2 )
     free( array[ i ] );
   hi->_pentry = NULL;
   printf( "   Size   Status\n" );
   printf( "   ----   ------\n" );
   while( _rtl_heapwalk( hi ) == _HEAPOK )
     printf( "%7u    %s\n", hi->_size, hi->_useflag ? "used" : "free" );
   free( hi );
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.1.6 calloc

Header File

alloc.h, stdlib.h

Category

Memory Routines

Prototype

```
void *calloc(size_t nitems, size_t size);
```

Description

Allocates main memory.

calloc provides access to the C memory heap. The heap is available for dynamic allocation of variable-sized blocks of memory. Many data structures, such as trees and lists, naturally employ heap memory allocation.

calloc allocates a block of size nitems * size. The block is initialized to 0.

Return Value

calloc returns a pointer to the newly allocated block. If not enough space exists for the new block or if nitems or size is 0, calloc returns NULL.

Example

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
   char *str = NULL;
   /* allocate memory for string */
   str = (char *) calloc(10, sizeof(char));
   /* copy "Hello" into string */
   strcpy(str, "Hello");
   /* display string */
   printf("String is %s\n", str);
   /* free memory */
   free(str);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.1.7 free

Header File

alloc.h, stdlib.h

Category

Memory Routines

Prototype

```
void free(void *block);
```

Description

Frees allocated block.

free deallocates a memory block allocated by a previous call to calloc, malloc, or realloc.

Return Value

None.

Example

```
#include <string.h>
#include <stdio.h>
#include <alloc.h>
int main(void)
{
   char *str;
   /* allocate memory for string */
   str = (char *) malloc(10);
   /* copy "Hello" to string */
   strcpy(str, "Hello");
   /* display string */
   printf("String is %s\n", str);
   /* free memory */
   free(str);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.1.8 heapcheck

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapcheck(void);
```

Description

Checks and verifies the heap.

heapcheck walks through the heap and examines each block, checking its pointers, size, and other critical attributes.

Return Value

The return value is less than 0 for an error and greater than 0 for success. The return values and their meaning are as follows:

_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_HEAPOK	Heap is verified

Example

```
#include <stdio.h>
#include <alloc.h>
#define NUM_PTRS  10
#define NUM_BYTES 16
int main(void)
{
   char *array[ NUM_PTRS ];
   int i;
   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );
   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );
   if( heapcheck() == _HEAPCORRUPT )
      printf( "Heap is corrupted.\n" );
   else
      printf( "Heap is OK.\n" );
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.1.9 heapcheckfree

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapcheckfree(unsigned int fillvalue);
```

Description

Checks the free blocks on the heap for a constant value.

Return Value

The return value is less then 0 for an error and greater than 0 for success. The return values and their meaning are as follows:

_BADVALUE	A value other than the fill value was found
_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_HEAPOK	Heap is accurate

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.1.10 heapchecknode

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapchecknode(void *node);
```

Description

Checks and verifies a single node on the heap.

If a node has been freed and heapchecknode is called with a pointer to the freed block, heapchecknode can return _BADNODE rather than the expected _FREEENTRY. This is because adjacent free blocks on the heap are merged, and the block in question no longer exists.

Return Value

One of the following values:

_BADNODE	Node could not be found
_FREEENTRY	Node is a free block
_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_USEDENTRY	Node is a used block

Example

```
#include <stdio.h>
#include <alloc.h>
#define NUM_PTRS  10
#define NUM_BYTES 16
int main(void)
{
   char *array[ NUM_PTRS ];
   int i;
   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );
   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );
   for( i = 0; i < NUM_PTRS; i++ )
   {
      printf( "Node %2d ", i );
      switch( heapchecknode( array[ i ] ) )
      {
         case _HEAPEMPTY:
              printf( "No heap.\n" );
              break;
         case _HEAPCORRUPT:
              printf( "Heap corrupt.\n" );
              break;
         case _BADNODE:
              printf( "Bad node.\n" );
              break;
         case _FREEENTRY:
```

```
                printf( "Free entry.\n" );
                break;
           case _USEDENTRY:
                printf( "Used entry.\n" );
                break;
           default:
                printf( "Unknown return code.\n" );
                 break;
        }
    }
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.1.11 heapfillfree

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapfillfree(unsigned int fillvalue);
```

Description

Fills the free blocks on the heap with a constant value.

Return Value

One of the following values:

_HEAPCORRUPT	Heap has been corrupted
_HEAPEMPTY	No heap
_HEAPOK	Heap is accurate

Example

```
#include <stdio.h>
#include <alloc.h>
#include <mem.h>
#define NUM_PTRS  10
#define NUM_BYTES 16
int main(void)
{
   char *array[ NUM_PTRS ];
   int i;
   int res;
   for( i = 0; i < NUM_PTRS; i++ )
      array[ i ] = (char *) malloc( NUM_BYTES );
   for( i = 0; i < NUM_PTRS; i += 2 )
      free( array[ i ] );
   if( heapfillfree( 1 ) < 0 )
   {
```

```
        printf( "Heap corrupted.\n" );
        return 1;
    }
    for( i = 1; i < NUM_PTRS; i += 2 )
        memset( array[ i ], 0, NUM_BYTES );
    res = heapcheckfree( 1 );
    if( res < 0 )
        switch( res )
        {
            case _HEAPCORRUPT:
                 printf( "Heap corrupted.\n" );
                 return 1;
            case _BADVALUE:
                 printf( "Bad value in free space.\n" );
                 return 1;
            default:
                 printf( "Unknown error.\n" );
                 return 1;
        }
    printf( "Test successful.\n" );
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.1.12 heapwalk

Header File

alloc.h

Category

Memory Routines

Prototype

```
int heapwalk(struct heapinfo *hi);
```

Description

heapwalk is used to “walk” through the heap, node by node.

heapwalk assumes the heap is correct. Use heapcheck to verify the heap before using heapwalk. _HEAPOK is returned with the last block on the heap. _HEAPEND will be returned on the next call to heapwalk.

heapwalk receives a pointer to a structure of type heapinfo (declared in alloc.h). For the first call to heapwalk, set the hi.ptr field to null. heapwalk returns with hi.ptr containing the address of the first block. hi.size holds the size of the block in bytes. hi.in_use is a flag that’s set if the block is currently in use.

Return Value

One of the following values:

_HEAPEMPTY	No heap exists
_HEAPEND	The end of the heap has been reached
_HEAPOK	The heapinfo block contains valid information about the next heap block

Example

```
Portability
<table htmltable<tr><th>POSIX</th><th>Win32</th><th>ANSI C</th><th>ANSI C++</th></tr><tr><td>
</td><td>+</td><td> </td><td> </td></tr></table
```

3.1.4.1.13 malloc

Header File

alloc.h, stdlib.h

Category

Memory Routines

Prototype

```
void *malloc(size_t size);
```

Description

malloc allocates a block of size bytes from the memory heap. It allows a program to allocate memory explicitly as it's needed, and in the exact amounts needed.

Allocates main memory.The heap is used for dynamic allocation of variable-sized blocks of memory. Many data structures, for example, trees and lists, naturally employ heap memory allocation.

In the large data models, all the space beyond the program stack to the end of available memory is available for the heap.

Return Value

On success, malloc returns a pointer to the newly allocated block of memory. If not enough space exists for the new block, it returns NULL. The contents of the block are left unchanged. If the argument size == 0, malloc returns NULL.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
int main(void)
{
    char *str;
    /* allocate memory for string */
    if ((str = (char *) malloc(10)) == NULL)
    {
       printf("Not enough memory to allocate buffer\n");
       exit(1);  /* terminate program if out of memory */
    }
    /* copy "Hello" into string */
    strcpy(str, "Hello");
    /* display string */
    printf("String is %s\n", str);
    /* free memory */
    free(str);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.1.14 realloc

Header File

alloc.h, stdlib.h

Category

Memory Routines

Prototype

```
void *realloc(void *block, size_t size);
```

Description

Reallocates main memory.

realloc attempts to shrink or expand the previously allocated block to size bytes. If size is zero, the memory block is freed and NULL is returned. The block argument points to a memory block previously obtained by calling malloc, calloc, or realloc. If block is a NULL pointer, realloc works just like malloc.

realloc adjusts the size of the allocated block to size, copying the contents to a new location if necessary.

Return Value

realloc returns the address of the reallocated block, which can be different than the address of the original block.

If the block cannot be reallocated, realloc returns NULL.

If the value of size is 0, the memory block is freed and realloc returns NULL.

Example

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
   char *str;
   /* allocate memory for string */
   str = (char *) malloc(10);
   /* copy "Hello" into string */
   strcpy(str, "Hello");
   printf("String is %s\n  Address is %p\n", str, str);
   str = (char *) realloc(str, 20);
   printf("String is %s\n  New address is %p\n", str, str);
   /* free memory */
   free(str);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.1.15 stackavail

Header File

malloc.h

Category

Memory Routines

Prototype

```
size_t stackavail(void);
```

Description

Gets the amount of available stack memory.

stackavail returns the number of bytes available on the stack. This is the amount of dynamic memory that alloca can access.

Return Value

stackavail returns a size_t value indicating the number of bytes available.

Example

```
#include <malloc.h>
#include <stdio.h>

int main(void)
{
  char *buf;

  printf("\nThe stack: %u\tstack pointer: %u", stackavail(), _SP);
  buf = (char *) alloca(100 * sizeof(char));
  printf("\nNow, the stack: %u\tstack pointer: %u", stackavail(), _SP);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.2 assert.h

The following functions, macros, and classes are provided in `assert.h`:

Topics

Name	Description
NDEBUG #define (see page 731)	**Header File** assert.h **Description** NDEBUG means "Use #define to treat assert as a macro or a true function". Can be defined in a user program. If defined, assert is a true function; otherwise assert is a macro.

assert (see page 731)	**Header File**
	assert.h
	Category
	Diagnostic Routines
	Prototype
	`void assert(int test);`
	Description
	Tests a condition and possibly aborts.
	assert is a macro that expands to an **if** statement; if test evaluates to zero, the assert macro calls the _assert function
	`void _RTLENTRY _EXPFUNC _assert(char * __cond, char * __file, int __line);`
	and aborts the program. The _assert function calls abort and asserts the following a message on stderr:
	`Assertion failed: test, file filename, line linenum`
	The filename and linenum listed in the message are the source file name and line number where the assert macro appears.
	If you place the #define... more (see page 731)

3.1.4.2.1 NDEBUG #define

Header File

assert.h

Description

NDEBUG means "Use #define to treat assert as a macro or a true function".

Can be defined in a user program. If defined, assert is a true function; otherwise assert is a macro.

3.1.4.2.2 assert

Header File

assert.h

Category

Diagnostic Routines

Prototype

```
void assert(int test);
```

Description

Tests a condition and possibly aborts.

assert is a macro that expands to an **if** statement; if test evaluates to zero, the assert macro calls the _assert function

```
void _RTLENTRY _EXPFUNC _assert(char * __cond, char * __file, int __line);
```

and aborts the program. The _assert function calls abort and asserts the following a message on stderr:

```
Assertion failed: test, file filename, line linenum
```

The filename and linenum listed in the message are the source file name and line number where the assert macro appears.

If you place the #define NDEBUG directive ("no debugging") in the source code before the #include <assert.h> directive, the macro expands to a no-op, the effect is to comment out the assert statement.

Return Value

None.

Example

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
struct ITEM {
   int key;
   int value;
};
/* add item to list, make sure list is not  null */
void additem(struct ITEM *itemptr) {
   assert(itemptr != NULL);
   /* add item to list */
}
int main(void)
{
   additem(NULL);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.3 conio.h

The following functions, macros, and classes are provided in `conio.h`:

Topics

Name	Description
_setcursortype (see page 740)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void _setcursortype(int cur_t);` **Description** Selects cursor appearance. Sets the cursor type to
cgets (see page 741)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `char *cgets(char *str);` **Description** Reads a string from the console. cgets reads a string of characters from the console, storing the string (and the string length) in the location pointed to by str. cgets reads characters until it encounters a carriage-return/linefeed (CR/LF) combination, or until the maximum allowable number of characters have been read. If cgets reads a CR/LF combination, it replaces the combination with a \0 (null terminator) before storing the string. Before cgets is called, set str[0] to the maximum length of the string to be read. On return, str[1]... more (see page 741)

clreol (see page 742)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void clreol(void);` **Description** Clears to end of line in text window. clreol clears all characters from the cursor position to the end of the line within the current text window, without moving the cursor. **Note:** This function should not be used in Win32 GUI applications. **Return Value** None. **Example**
clrscr (see page 743)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void clrscr(void);` **Description** Clears the text-mode window. clrscr clears the current text window and places the cursor in the upper left corner (at position 1,1). **Note:** Do not use this function in Win32 GUI applications. **Return Value** None. **Example**
cprintf (see page 744)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int cprintf(const char *format[, argument, ...]);` **Description** Writes formatted output to the screen. cprintf accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data directly to the current text window on the screen. There must be the same number of format specifiers as arguments. For details details on format specifiers, see printf Format Specifiers. The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global... more (see page 744)
cputs (see page 745)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int cputs(const char *str);` **Description** Writes a string to the screen. cputs writes the null-terminated string str to the current text window. It does not append a newline character. The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable _directvideo. Unlike puts, cputs does not translate linefeed characters (\n) into carriage-return/linefeed character pairs (\r\n). **Note:** Do not use this function in Win32 GUI applications. **Return Value** cputs returns the last character printed. **Example**

cscanf (see page 746)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int cscanf(char *format[, address, ...]);` **Description** Scans and formats input from the console. cscanf scans a series of input fields one character at a time, reading directly from the console. Then each field is formatted according to a format specifier passed to cscanf in the format string pointed to by format. Finally, cscanf stores the formatted input at an address passed to it as an argument following format, and echoes the input directly to the screen. There must be the same number of format specifiers and addresses as there are input fields.... more (see page 746)
delline (see page 747)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void delline(void);` **Description** Deletes line in text window. delline deletes the line containing the cursor and moves all lines below it one line up. delline operates within the currently active text window. **Note:** Do not use this function in Win32 GUI applications. **Return Value** None. **Example**
getch (see page 747)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int getch(void);` **Description** Gets character from keyboard, does not echo to screen. getch reads a single character directly from the keyboard, without echoing to the screen. **Note:** Do not use this function in Win32 GUI applications. **Return Value** getch returns the character read from the keyboard. **Example**
getche (see page 748)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int getche(void);` **Description** Gets character from the keyboard, echoes to screen. getche reads a single character from the keyboard and echoes it to the current text window using direct video or BIOS. **Note:** Do not use this function in Win32 GUI applications. **Return Value** getche returns the character read from the keyboard. **Example**

getpass (see page 749)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `char *getpass(const char *prompt);` **Description** Reads a password. getpass reads a password from the system console after prompting with the null-terminated string prompt and disabling the echo. A pointer is returned to a null-terminated string of up to eight characters (not counting the null-terminator). **Note:** Do not use this function in Win32 GUI applications. **Return Value** The return value is a pointer to a static string that is overwritten with each call. **Example**
gettext (see page 750)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int gettext(int left, int top, int right, int bottom, void *destin);` **Description** Copies text from text mode screen to memory. gettext stores the contents of an onscreen text rectangle defined by left, top, right, and bottom into the area of memory pointed to by destin. All coordinates are absolute screen coordinates not window-relative. The upper left corner is (1,1). gettext reads the contents of the rectangle into memory sequentially from left to right and top to bottom. Each position onscreen takes 2 bytes of memory: The first byte is the character in... more (see page 750)

gettextinfo (see page 751)	**Header File**
	conio.h
	Category
	Console I/O Routines
	Prototype
	`void gettextinfo(struct text_info *r);`
	Description
	Gets text mode video information.
	gettextinfo fills in the text_info structure pointed to by r with the current text video information.
	The text_info structure is defined in conio.h as follows:
	`struct text_info {`
	`unsigned char winleft; /* left window coordinate */`
	`unsigned char wintop; /* top window coordinate */`
	`unsigned char winright; /* right window coordinate */`
	`unsigned char winbottom; /* bottom window coordinate */`
	`unsigned char attribute; /* text attribute */`
	`unsigned char normattr; /* normal attribute */`
	`unsigned char currmode; /* BW40, BW80, C40, C80,...` more (see page 751)

gotoxy (see page 752)	**Header File** conio.h **Category** Console I/O Routines **Prototype** void gotoxy(int x, int y); **Description** Positions cursor in text window. gotoxy moves the cursor to the given position in the current text window. If the coordinates are in any way invalid the call to gotoxy is ignored. An example of this is a call to gotoxy(40,30) when (35,25) is the bottom right position in the window. Neither argument to gotoxy can be zero. **Note:**Do not use this function in Win32 GUI applications. **Return Value** None. **Example**
highvideo (see page 752)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void highvideo(void);` **Description** Selects high-intensity characters. highvideo selects high-intensity characters by setting the high-intensity bit of the currently selected foreground color. This function does not affect any characters currently onscreen, but does affect those displayed by functions (such as cprintf) that perform direct video, text mode output after highvideo is called. **Note:** Do not use this function in Win32 GUI applications. **Return Value** None. **Example**
insline (see page 753)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void insline(void);` **Description** Inserts a blank line in the text window. insline inserts an empty line in the text window at the cursor position using the current text background color. All lines below the empty one move down one line, and the bottom line scrolls off the bottom of the window. **Note:** Do not use this function in Win32 GUI applications. **Return Value** None. **Example**
kbhit (see page 754)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int kbhit(void);` **Description** Checks for currently available keystrokes. kbhit checks to see if a keystroke is currently available. Any available keystrokes can be retrieved with getch or getche. **Note:** Do not use this function in Win32 GUI applications. **Return Value** If a keystroke is available, kbhit returns a nonzero value. Otherwise, it returns 0. **Example**

lowvideo (see page 755)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void lowvideo(void);` **Description** Selects low-intensity characters. lowvideo selects low-intensity characters by clearing the high-intensity bit of the currently selected foreground color. This function does not affect any characters currently onscreen. It affects only those characters displayed by functions that perform text mode, direct console output after this function is called. **Note:** Do not use this function in Win32 GUI applications. **Return Value** None. **Example**
movetext (see page 755)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int movetext(int left, int top, int right, int bottom, int destleft, int desttop);` **Description** Copies text onscreen from one rectangle to another. movetext copies the contents of the onscreen rectangle defined by left, top, right, and bottom to a new rectangle of the same dimensions. The new rectangle's upper left corner is position (destleft, desttop). All coordinates are absolute screen coordinates. Rectangles that overlap are moved correctly. movetext is a text mode function performing direct video output. **Note:**Do not use this function in Win32 GUI applications. **Return Value** On success, movetext... more (see page 755)
normvideo (see page 756)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void normvideo(void);` **Description** Selects normal-intensity characters. normvideo selects normal characters by returning the text attribute (foreground and background) to the value it had when the program started. This function does not affect any characters currently on the screen, only those displayed by functions (such as cprintf) performing direct console output functions after normvideo is called. **Note:** Do not use this function in Win32 GUI applications. **Return Value** None. **Example**

putch (see page 757)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int putch(int c);` **Description** Outputs character to screen. putch outputs the character c to the current text window. It is a text mode function performing direct video output to the console. putch does not translate linefeed characters (\n) into carriage-return/linefeed pairs. The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable _directvideo. **Note:** This function should not be used in Win32 GUI applications. **Return Value** On success, putch returns the character printed, c. On error, it returns EOF.... more (see page 757)
puttext (see page 758)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int puttext(int left, int top, int right, int bottom, void *source);` **Description** Copies text from memory to the text mode screen. puttext writes the contents of the memory area pointed to by source out to the onscreen rectangle defined by left, top, right, and bottom. All coordinates are absolute screen coordinates, not window-relative. The upper left corner is (1,1). puttext places the contents of a memory area into the defined rectangle sequentially from left to right and top to bottom. Each position onscreen takes 2 bytes of memory: The first byte is... more (see page 758)
textattr (see page 759)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void textattr(int newattr);` **Description** Sets text attributes. **Note:** Do not use this function in Win32 GUI applications. textattr lets you set both the foreground and background colors in a single call. (Normally, you set the attributes with textcolor and textbackground.) This function does not affect any characters currently onscreen; it affects only those characters displayed by functions (such as cprintf) performing text mode, direct video output after this function is called. The color information is encoded in the newattr parameter as follows: In this 8-bit newattr parameter: • bits 0-3 contain the 4-bit... more (see page 759)

textbackground (see page 760)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void textbackground(int newcolor);` **Description** Selects new text background color. **Note:** Do not use this function in Win32 GUI applications. textbackground selects the background color. This function works for functions that produce output in text mode directly to the screen. newcolor selects the new background color. You can set newcolor to an integer from 0 to 7, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h. Once you have called textbackground, all subsequent functions using direct video output (such as cprintf) will use... more (see page 760)
textcolor (see page 761)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void textcolor(int newcolor);` **Description** Selects new character color in text mode. **Note:** Do not use this function in Win32 GUI applications. textcolor selects the foreground character color. This function works for the console output functions. newcolor selects the new foreground color. You can set newcolor to an integer as given in the table below, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h. Once you have called textcolor, all subsequent functions using direct video output (such as cprintf) will use newcolor. textcolor... more (see page 761)
textmode (see page 762)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void textmode(int newmode);` **Description** Puts screen in text mode. **Note:** Do not use this function in Win32 GUI applications. textmode selects a specific text mode. You can give the text mode (the argument newmode) by using a symbolic constant from the enumeration type text_modes (defined in conio.h). The most commonly used text_modes type constants and the modes they specify are given in the following table. Some additional values are defined in conio.h.
ungetch (see page 764)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int ungetch(int ch);` **Description** Pushes a character back to the keyboard buffer. **Note:** Do not use this function in Win32 GUI applications. ungetch pushes the character ch back to the console, causing ch to be the next character read. The ungetch function fails if it is called more than once before the next read. **Return Value** On success, ungetch returns the character ch. On error, it returns EOF. **Example**

wherex (see page 764)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int wherex(void);` **Description** Gives horizontal cursor position within window. **Note:** Do not use this function in Win32 GUI applications. wherex returns the x-coordinate of the current cursor position (within the current text window). **Return Value** wherex returns an integer in the range 1 to the number of columns in the current video mode. **Example**
wherey (see page 765)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `int wherey(void);` **Description** Gives vertical cursor position within window. **Note:** Do not use this function in Win32 GUI applications. wherey returns the y-coordinate of the current cursor position (within the current text window). **Return Value** wherey returns an integer in the range 1 to the number of rows in the current video mode. **Portability**
window (see page 766)	**Header File** conio.h **Category** Console I/O Routines **Prototype** `void window(int left, int top, int right, int bottom);` **Description** Defines active text mode window. **Note:** Do not use this function in Win32 GUI applications. window defines a text window onscreen. If the coordinates are in any way invalid, the call to window is ignored. left and top are the screen coordinates of the upper left corner of the window. right and bottom are the screen coordinates of the lower right corner. The minimum size of the text window is one column by one line. The default window is full screen, with... more (see page 766)

3.1.4.3.1 _setcursortype

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void _setcursortype(int cur_t);
```

Description

Selects cursor appearance.

Sets the cursor type to

_NOCURSOR	Turns off the cursor
_NORMALCURSOR	Normal underscore cursor
_SOLIDCURSOR	Solid block cursor

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main( )
{
  // tell the user what to do
  clrscr();
  cputs("Press any key three times.\n\r");
  cputs("Each time the cursor will change shape.\n\r");
  gotoxy(1,5);           // show a solid cursor
  cputs("Now the cursor is solid.\n\r");
  _setcursortype(_SOLIDCURSOR);
  while(!kbhit()) {};         // wait to proceed
  getch();
  gotoxy(1,5);           // remove the cursor
  cputs("Now the cursor is gone.");
  clreol();
  gotoxy(1,6);
  _setcursortype(_NOCURSOR);
  while(!kbhit()) {};         // wait to proceed
  getch();
  gotoxy(1,5);                 // show a normal cursor
  cputs("Now the cursor is normal.");
  clreol();
  gotoxy(1,6);
  _setcursortype(_NORMALCURSOR);
  while(!kbhit()) {};         // wait to proceed
  getch();
  clrscr();
  return(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.2 cgets

Header File

conio.h

Category

Console I/O Routines

Prototype

```
char *cgets(char *str);
```

Description

Reads a string from the console.

cgets reads a string of characters from the console, storing the string (and the string length) in the location pointed to by str.

cgets reads characters until it encounters a carriage-return/linefeed (CR/LF) combination, or until the maximum allowable number of characters have been read. If cgets reads a CR/LF combination, it replaces the combination with a \0 (null terminator) before storing the string.

Before cgets is called, set str[0] to the maximum length of the string to be read. On return, str[1] is set to the number of characters actually read. The characters read start at str[2] and end with a null terminator. Thus, str must be at least str[0] plus 2 bytes long.

Note: Do not use this function for Win32 GUI applications.

Return Value

On success, cgets returns a pointer to str[2].

Example

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
   char buffer[83];
   char *p;
   /* There is space for 80 characters plus the NULL terminator */
   buffer[0] = 81;
   printf("Input some chars:");
   p = cgets(buffer);
   printf("\ncgets read %d characters: \"%s\"\n", buffer[1], p);
   printf("The returned pointer is %p, buffer[0] is at %p\n", p, &buffer);
   /* Leave room for 5 characters plus the NULL terminator */
   buffer[0] = 6;
   printf("Input some chars:");
   p = cgets(buffer);
   printf("\ncgets read %d characters: \"%s\"\n", buffer[1], p);
   printf("The returned pointer is %p, buffer[0] is at %p\n", p, &buffer);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.3 clreol

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void clreol(void);
```

Description

Clears to end of line in text window.

clreol clears all characters from the cursor position to the end of the line within the current text window, without moving the cursor.

Note: This function should not be used in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   clrscr();
   cprintf("The function CLREOL clears all characters from the\r\n");
   cprintf("cursor position to the end of the line within the\r\n");
   cprintf("current text window, without moving the cursor.\r\n");
   cprintf("Press any key to continue . . .");
   gotoxy(14, 4);
   getch();
   clreol();
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.4 clrscr

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void clrscr(void);
```

Description

Clears the text-mode window.

clrscr clears the current text window and places the cursor in the upper left corner (at position 1,1).

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   int i;
   clrscr();
```

```
    for (i = 0; i < 20; i++)
       cprintf("%d\r\n", i);
    cprintf("\r\nPress any key to clear screen");
    getch();
    clrscr();
    cprintf("The screen has been cleared!");
    getch();
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.5 cprintf

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int cprintf(const char *format[, argument, ...]);
```

Description

Writes formatted output to the screen.

cprintf accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data directly to the current text window on the screen. There must be the same number of format specifiers as arguments.

For details details on format specifiers, see printf Format Specifiers.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable _directvideo.

Unlike fprintf and printf, cprintf does not translate linefeed characters (\n) into carriage-return/linefeed character pairs (\r\n). Tab characters (specified by \t) are not expanded into spaces.

Note: Do not use this function in Win32 GUI applications.

Return Value

cprintf returns the number of characters output.

Example

```
#include <conio.h>
int main(void)
{
   /* clear the screen */
   clrscr();
   /* create a text window */
   window(10, 10, 80, 25);
   /* output some text in the window */
   cprintf("Hello world\r\n");
   /* wait for a key */
```

```
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.6 cputs

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int cputs(const char *str);
```

Description

Writes a string to the screen.

cputs writes the null-terminated string str to the current text window. It does not append a newline character.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable _directvideo. Unlike puts, cputs does not translate linefeed characters (\n) into carriage-return/linefeed character pairs (\r\n).

Note: Do not use this function in Win32 GUI applications.

Return Value

cputs returns the last character printed.

Example

```
#include <conio.h>
int main(void)
{
   /* clear the screen */
   clrscr();
   /* create a text window */
   window(10, 10, 80, 25);
   /* output some text in the window */
   cputs("This is within the window\r\n");
   /* wait for a key */
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.7 cscanf

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int cscanf(char *format[, address, ...]);
```

Description

Scans and formats input from the console.

cscanf scans a series of input fields one character at a time, reading directly from the console. Then each field is formatted according to a format specifier passed to cscanf in the format string pointed to by format. Finally, cscanf stores the formatted input at an address passed to it as an argument following format, and echoes the input directly to the screen. There must be the same number of format specifiers and addresses as there are input fields.

Note: For details on format specifiers, see scanf Format Specifiers.

cscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely for a number of reasons. See scanf for a discussion of possible causes.

Note: Do not use this function in Win32 GUI applications.

Return Value

cscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If cscanf attempts to read at end-of-file , the return value is EOF.

Example

```
#include <conio.h>
int main(void)
{
   char string[80];
   /* clear the screen */
   clrscr();
   /* Prompt the user for input */
   cprintf("Enter a string with no spaces:");
   /* read the input */
   cscanf("%s", string);
   /* display what was read */
   cprintf("\r\nThe string entered is: %s", string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.8 delline

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void delline(void);
```

Description

Deletes line in text window.

delline deletes the line containing the cursor and moves all lines below it one line up. delline operates within the currently active text window.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   clrscr();
   cprintf("The function DELLINE deletes the line containing the\r\n");
   cprintf("cursor and moves all lines below it one line up.\r\n");
   cprintf("DELLINE operates within the currently active text\r\n");
   cprintf("window.  Press any key to continue . . .");
   gotoxy(1,2);  /* Move the cursor to the second line and first column */
   getch();
   delline();
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.9 getch

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int getch(void);
```

Description

Gets character from keyboard, does not echo to screen.

getch reads a single character directly from the keyboard, without echoing to the screen.

Note: Do not use this function in Win32 GUI applications.

Return Value

getch returns the character read from the keyboard.

Example

```
#include <conio.h>
#include <stdio.h>
int main(void)
{
  int c;
  int extended = 0;
  c = getch();
  if (!c)
    extended = getch();
  if (extended)
    printf("The character is extended\n");
  else
    printf("The character isn't extended\n");
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.10 getche

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int getche(void);
```

Description

Gets character from the keyboard, echoes to screen.

getche reads a single character from the keyboard and echoes it to the current text window using direct video or BIOS.

Note: Do not use this function in Win32 GUI applications.

Return Value

getche returns the character read from the keyboard.

Example

```
#include <stdio.h>
#include <conio.h>
```

```
int main(void)
{
   char ch;
   printf("Input a character:");
   ch = getche();
   printf("\nYou input a '%c'\n", ch);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.11 getpass

Header File

conio.h

Category

Console I/O Routines

Prototype

```
char *getpass(const char *prompt);
```

Description

Reads a password.

getpass reads a password from the system console after prompting with the null-terminated string prompt and disabling the echo. A pointer is returned to a null-terminated string of up to eight characters (not counting the null-terminator).

Note: Do not use this function in Win32 GUI applications.

Return Value

The return value is a pointer to a static string that is overwritten with each call.

Example

```
#include <conio.h>

int main(void)
{
   char *password;

   password = getpass("Input a password:");
   cprintf("The password is: %s\r\n", password);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.12 gettext

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int gettext(int left, int top, int right, int bottom, void *destin);
```

Description

Copies text from text mode screen to memory.

gettext stores the contents of an onscreen text rectangle defined by left, top, right, and bottom into the area of memory pointed to by destin.

All coordinates are absolute screen coordinates not window-relative. The upper left corner is (1,1). gettext reads the contents of the rectangle into memory sequentially from left to right and top to bottom.

Each position onscreen takes 2 bytes of memory: The first byte is the character in the cell and the second is the cell's video attribute. The space required for a rectangle w columns wide by h rows high is defined as

bytes = (h rows) x (w columns) x 2

Note: Do not use this function in Win32 GUI applications.

Return Value

gettext returns 1 if the operation succeeds.

On error, it returns 0 (for example, if it fails because you gave coordinates outside the range of the current screen mode).

Example

```
#include <conio.h>

char buffer[4096];
int main(void)
{
   int i;
   clrscr();
   for (i = 0; i <= 20; i++)
      cprintf("Line #%d\r\n", i);
   gettext(1, 1, 80, 25, buffer);
   gotoxy(1, 25);
   cprintf("Press any key to clear screen...");
   getch();
   clrscr();
   gotoxy(1, 25);
   cprintf("Press any key to restore screen...");
   getch();
   puttext(1, 1, 80, 25, buffer);
   gotoxy(1, 25);
   cprintf("Press any key to quit...");
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.13 gettextinfo

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void gettextinfo(struct text_info *r);
```

Description

Gets text mode video information.

gettextinfo fills in the text_info structure pointed to by r with the current text video information.

The text_info structure is defined in conio.h as follows:

```
struct text_info {
unsigned char winleft; /* left window coordinate */
unsigned char wintop; /* top window coordinate */
unsigned char winright; /* right window coordinate */
unsigned char winbottom; /* bottom window coordinate */
unsigned char attribute; /* text attribute */
unsigned char normattr; /* normal attribute */
unsigned char currmode; /* BW40, BW80, C40, C80, or C4350 */
unsigned char screenheight; /* text screen's height */
unsigned char screenwidth; /* text screen's width */
unsigned char curx; /* x-coordinate in current window */
unsigned char cury; /* y-coordinate in current window */
};
```

Note: Do not use this function in Win32 GUI applications.

Return Value

None. Results are returned in the structure pointed to by r.

Example

```
#include <conio.h>
int main(void)
 {
   struct text_info ti;
   gettextinfo(&ti);
   cprintf("window left      %2d\r\n",ti.winleft);
   cprintf("window top       %2d\r\n",ti.wintop);
```

```
    cprintf("window right     %2d\r\n",ti.winright);
    cprintf("window bottom    %2d\r\n",ti.winbottom);
    cprintf("attribute        %2d\r\n",ti.attribute);
    cprintf("normal attribute %2d\r\n",ti.normattr);
    cprintf("current mode     %2d\r\n",ti.currmode);
    cprintf("screen height    %2d\r\n",ti.screenheight);
    cprintf("screen width     %2d\r\n",ti.screenwidth);
    cprintf("current x        %2d\r\n",ti.curx);
    cprintf("current y        %2d\r\n",ti.cury);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.14 gotoxy

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void gotoxy(int x, int y);
```

Description

Positions cursor in text window.

gotoxy moves the cursor to the given position in the current text window. If the coordinates are in any way invalid the call to gotoxy is ignored. An example of this is a call to gotoxy(40,30) when (35,25) is the bottom right position in the window. Neither argument to gotoxy can be zero.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
    clrscr();
    gotoxy(35, 12);
    cprintf("Hello world");
    getch();
    return 0;
}
```

3.1.4.3.15 highvideo

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void highvideo(void);
```

Description

Selects high-intensity characters.

highvideo selects high-intensity characters by setting the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently onscreen, but does affect those displayed by functions (such as cprintf) that perform direct video, text mode output after highvideo is called.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
clrscr();
lowvideo();
cprintf("Low Intensity text\r\n");
highvideo();
gotoxy(1,2);
cprintf("High Intensity Text\r\n");
return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.16 insline

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void insline(void);
```

Description

Inserts a blank line in the text window.

insline inserts an empty line in the text window at the cursor position using the current text background color. All lines below the empty one move down one line, and the bottom line scrolls off the bottom of the window.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
    clrscr();
    cprintf("INSLINE inserts an empty line in the text window\r\n");
 cprintf("at the cursor position using the current text\r\n");
    cprintf("background color.  All lines below the empty one\r\n");
    cprintf("move down one line and the bottom line scrolls\r\n");
 cprintf("off the bottom of the window.\r\n");
    cprintf("\r\nPress any key to continue:");
 gotoxy(1, 3);
 getch();
    insline();
    getch();
 return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.17 kbhit

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int kbhit(void);
```

Description

Checks for currently available keystrokes.

kbhit checks to see if a keystroke is currently available. Any available keystrokes can be retrieved with getch or getche.

Note: Do not use this function in Win32 GUI applications.

Return Value

If a keystroke is available, kbhit returns a nonzero value. Otherwise, it returns 0.

Example

```
#include <conio.h>
int main(void)
{
 cprintf("Press any key to continue:");
    while (!kbhit()) /* do nothing */ ;
    cprintf("\r\nA key was pressed...\r\n");
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.18 lowvideo

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void lowvideo(void);
```

Description

Selects low-intensity characters.

lowvideo selects low-intensity characters by clearing the high-intensity bit of the currently selected foreground color.

This function does not affect any characters currently onscreen. It affects only those characters displayed by functions that perform text mode, direct console output after this function is called.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
clrscr();
highvideo();
   cprintf("High Intensity Text\r\n");
   lowvideo();
gotoxy(1,2);
   cprintf("Low Intensity Text\r\n");
return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.19 movetext

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int movetext(int left, int top, int right, int bottom, int destleft, int desttop);
```

Description

Copies text onscreen from one rectangle to another.

movetext copies the contents of the onscreen rectangle defined by left, top, right, and bottom to a new rectangle of the same dimensions. The new rectangle's upper left corner is position (destleft, desttop).

All coordinates are absolute screen coordinates. Rectangles that overlap are moved correctly.

movetext is a text mode function performing direct video output.

Note: Do not use this function in Win32 GUI applications.

Return Value

On success, movetext returns nonzero.

On error (for example, if it failed because you gave coordinates outside the range of the current screen mode), movetext returns 0.

Example

```
#include <conio.h>
#include <string.h>
int main(void)
{
    char *str = "This is a test string";
    clrscr();
    cputs(str);
    getch();
    movetext(1, 1, strlen(str), 2, 10, 10);
    getch();
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.20 normvideo

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void normvideo(void);
```

Description

Selects normal-intensity characters.

normvideo selects normal characters by returning the text attribute (foreground and background) to the value it had when the program started.

This function does not affect any characters currently on the screen, only those displayed by functions (such as cprintf) performing direct console output functions after normvideo is called.

Note: Do not use this function in Win32 GUI applications.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
    normvideo();
    cprintf("NORMAL Intensity Text\r\n");
 return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.21 putch

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int putch(int c);
```

Description

Outputs character to screen.

putch outputs the character c to the current text window. It is a text mode function performing direct video output to the console. putch does not translate linefeed characters (\n) into carriage-return/linefeed pairs.

The string is written either directly to screen memory or by way of a BIOS call, depending on the value of the global variable _directvideo.

Note: This function should not be used in Win32 GUI applications.

Return Value

On success, putch returns the character printed, c. On error, it returns EOF.

Example

```
#include <stdio.h>
#include <conio.h>
int main(void)
{
```

```
  char ch = 0;
  printf("Input a string:");
  while ((ch != '\r'))
  {
    ch = getch();
    putch(ch);
  }
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.22 puttext

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int puttext(int left, int top, int right, int bottom, void *source);
```

Description

Copies text from memory to the text mode screen.

puttext writes the contents of the memory area pointed to by source out to the onscreen rectangle defined by left, top, right, and bottom.

All coordinates are absolute screen coordinates, not window-relative. The upper left corner is (1,1).

puttext places the contents of a memory area into the defined rectangle sequentially from left to right and top to bottom.

Each position onscreen takes 2 bytes of memory: The first byte is the character in the cell, and the second is the cell's video attribute. The space required for a rectangle w columns wide by h rows high is defined as

bytes = (h rows) x (w columns) x 2

puttext is a text mode function performing direct video output.

Note: This function should not be used in Win32 GUI applications.

Return Value

puttext returns a nonzero value if the operation succeeds; it returns 0 if it fails (for example, if you gave coordinates outside the range of the current screen mode).

Example

```
#include <conio.h>
int main(void)
{
   char buffer[512];
   /* put some text to the console */
   clrscr();
   gotoxy(20, 12);
   cprintf("This is a test.  Press any key to continue ...");
```

```
    getch();
    /* grab screen contents */
    gettext(20, 12, 36, 21,buffer);
    clrscr();
    /* put selected characters back to the screen */
    gotoxy(20, 12);
    puttext(20, 12, 36, 21, buffer);
    getch();
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.23 textattr

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void textattr(int newattr);
```

Description

Sets text attributes.

Note: Do not use this function in Win32 GUI applications.

textattr lets you set both the foreground and background colors in a single call. (Normally, you set the attributes with textcolor and textbackground.)

This function does not affect any characters currently onscreen; it affects only those characters displayed by functions (such as cprintf) performing text mode, direct video output after this function is called.

The color information is encoded in the newattr parameter as follows:

In this 8-bit newattr parameter:

- bits 0-3 contain the 4-bit foreground color (0 to 15).
- bits 4-6 contain the 3-bit background color (0 to 7).
- bit 8 is the blink-enable bit.

If the blink-enable bit is on, the character blinks. This can be accomplished by adding the constant BLINK to the attribute.

- If you use the symbolic color constants defined in conio.h for creating text attributes with textattr, note the following limitations on the color you select for the background:
- You can select only one of the first eight colors for the background.
- You must shift the selected background color left by 4 bits to move it into the correct bit positions.

These symbolic constants are listed in the following table:

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   int i;
   clrscr();
   for (i=0; i<9; i++)
   {
       textattr(i + ((i+1) << 4));
       cprintf("This is a test\r\n");
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.24 textbackground

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void textbackground(int newcolor);
```

Description

Selects new text background color.

Note: Do not use this function in Win32 GUI applications.

textbackground selects the background color. This function works for functions that produce output in text mode directly to the screen. newcolor selects the new background color. You can set newcolor to an integer from 0 to 7, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h.

Once you have called textbackground, all subsequent functions using direct video output (such as cprintf) will use newcolor. textbackground does not affect any characters currently onscreen.

The following table lists the symbolic constants and the numeric values of the allowable colors:

BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   int i, j;
   clrscr();
   for (i=0; i<9; i++)
   {
       for (j=0; j<80; j++)
          cprintf("C");
       cprintf("\r\n");
       textcolor(i+1);
       textbackground(i);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.25 textcolor

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void textcolor(int newcolor);
```

Description

Selects new character color in text mode.

Note: Do not use this function in Win32 GUI applications.

textcolor selects the foreground character color. This function works for the console output functions. newcolor selects the new foreground color. You can set newcolor to an integer as given in the table below, or to one of the symbolic constants defined in conio.h. If you use symbolic constants, you must include conio.h.

Once you have called textcolor, all subsequent functions using direct video output (such as cprintf) will use newcolor. textcolor does not affect any characters currently onscreen.

The following table lists the allowable colors (as symbolic constants) and their numeric values:

BLACK	0
BLUE	1
GREEN	2
CYAN	3

RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15
BLINK	128

You can make the characters blink by adding 128 to the foreground color. The predefined constant BLINK exists for this purpose.

For example:

```
textcolor(CYAN + BLINK);
```

Note: Some monitors do not recognize the intensity signal used to create the eight “light” colors (8-15). On such monitors, the light colors are displayed as their “dark” equivalents (0-7). Also, systems that do not display in color can treat these numbers as shades of one color, special patterns, or special attributes (such as underlined, bold, italics, and so on). Exactly what you’ll see on such systems depends on your hardware.

Return Value

None.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.26 textmode

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void textmode(int newmode);
```

Description

Puts screen in text mode.

Note: Do not use this function in Win32 GUI applications.

textmode selects a specific text mode.

You can give the text mode (the argument newmode) by using a symbolic constant from the enumeration type text_modes (defined in conio.h).

The most commonly used text_modes type constants and the modes they specify are given in the following table. Some additional values are defined in conio.h.

LASTMODE	Previous text mode
BW40	Black and white, 40 columns
C40	Color, 40 columns
BW80	Black and white, 80 columns
C80	Color, 80 columns
MONO	Monochrome, 80 columns
C4350	EGA 43-line and VGA 50-line modes

When textmode is called, the current window is reset to the entire screen, and the current text attributes are reset to normal, corresponding to a call to normvideo.

Specifying LASTMODE to textmode causes the most recently selected text mode to be reselected.

textmode should be used only when the screen or window is in text mode (presumably to change to a different text mode). This is the only context in which textmode should be used. When the screen is in graphics mode, use restorecrtmode instead to escape temporarily to text mode.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   textmode(BW40);
   cprintf("ABC");
   getch();
   textmode(C40);
   cprintf("ABC");
   getch();
   textmode(BW80);
   cprintf("ABC");
   getch();
   textmode(C80);
   cprintf("ABC");
   getch();
   textmode(MONO);
   cprintf("ABC");
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.27 ungetch

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int ungetch(int ch);
```

Description

Pushes a character back to the keyboard buffer.

Note: Do not use this function in Win32 GUI applications.

ungetch pushes the character ch back to the console, causing ch to be the next character read. The ungetch function fails if it is called more than once before the next read.

Return Value

On success, ungetch returns the character ch.

On error, it returns EOF.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
int main( void )
{
   int i=0;
   char ch;
   puts("Input an integer followed by a char:");
   /* read chars until non digit or EOF */
   while((ch = getche()) != EOF && isdigit(ch))
      i = 10 * i + ch - 48; /* convert ASCII into int value */
   /* if non digit char was read, push it back into input buffer */
   if (ch != EOF)
      ungetch(ch);
   printf("\n\ni = %d, next char in buffer = %c\n", i, getch());
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.28 wherex

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int wherex(void);
```

Description

Gives horizontal cursor position within window.

Note: Do not use this function in Win32 GUI applications.

wherex returns the x-coordinate of the current cursor position (within the current text window).

Return Value

wherex returns an integer in the range 1 to the number of columns in the current video mode.

Example

```
#include <conio.h>
int main(void)
{
   clrscr();
   gotoxy(10,10);
   cprintf("Current location is X: %d  Y: %d\r\n", wherex(), wherey());
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.29 wherey

Header File

conio.h

Category

Console I/O Routines

Prototype

```
int wherey(void);
```

Description

Gives vertical cursor position within window.

Note: Do not use this function in Win32 GUI applications.

wherey returns the y-coordinate of the current cursor position (within the current text window).

Return Value

wherey returns an integer in the range 1 to the number of rows in the current video mode.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.3.30 window

Header File

conio.h

Category

Console I/O Routines

Prototype

```
void window(int left, int top, int right, int bottom);
```

Description

Defines active text mode window.

Note: Do not use this function in Win32 GUI applications.

window defines a text window onscreen. If the coordinates are in any way invalid, the call to window is ignored.

left and top are the screen coordinates of the upper left corner of the window.

right and bottom are the screen coordinates of the lower right corner.

The minimum size of the text window is one column by one line. The default window is full screen, with the coordinates:

1,1,C,R

where C is the number of columns in the current video mode, and R is the number of rows.

Return Value

None.

Example

```
#include <conio.h>
int main(void)
{
   window(10,10,40,11);
   textcolor(BLACK);
   textbackground(WHITE);
   cprintf("This is a test\r\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.4 ctype.h

The following functions, macros, and classes are provided in `ctype.h`:

Topics

Name	Description
_ctype (see page 772)	**Header File** ctype.h **Syntax** `extern char _ctype[];` **Description** _ctype is an array of character attribute information indexed by ASCII value + 1. Each entry is a set of bits describing the character. This array is used by isdigit, isprint, and so on.
_IS_xxx #defines (see page 772)	**Header File** ctype.h **Description** Bit settings in the _ctype[] used by the is... character macros.
_tolower (see page 772)	**Header File** ctype.h **Category** Conversion Routines **Prototype** `int _tolower(int ch);` **Description** _tolower is a macro that does the same conversion as tolower, except that it should be used only when ch is known to be uppercase (AZ). To use _tolower, you must include ctype.h. **Return Value** _tolower returns the converted value of ch if it is uppercase; otherwise, the result is undefined. **Example**
_toupper (see page 773)	**Header File** ctype.h **Category** Conversion Routines **Prototype** `int _toupper(int ch);` **Description** Translates characters to uppercase. _toupper is a macro that does the same conversion as toupper, except that it should be used only when ch is known to be lowercase (a to z). To use _toupper, you must include ctype.h. **Return Value** _toupper returns the converted value of ch if it is lowercase; otherwise, the result is undefined. **Example**

isalnum, __iscsym, iswalnum, _ismbcalnum (see page 774)	**Header File**
	ctype.h, mbstring.h
	Category
	Classification Routines
	Prototype
	`int isalnum(int c);`
	`int __iscsym(int c);`
	`int iswalnum(wint_t c);`
	`int _ismbcalnum(unsigned int c);`
	Description
	Tests for an alphanumeric character.
	isalnum is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a letter (A to Z or a to z) or a digit (0 to 9).
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	It is a predicate returning nonzero for true and 0 for false.... more (see page 774)

isalpha, __iscsymf, iswalpha, _ismbcalpha (see page 775)	**Header File**
	ctype.h, mbstring.h
	Category
	Classification Routines
	Prototype
	`int isalpha(int c);`
	`int __iscsymf(int c);`
	`int iswalpha(wint_t c);`
	`int _ismbcalpha(unsigned int c);`
	Description
	Classifies an alphabetical character.
	isalpha is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a letter (A to Z or a to z).
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	isalpha returns nonzero if c is a letter.
	__iscsymf returns true if and only if the argument c is... more (see page 775)

isascii, iswascii (see page 776)	**Header File**
	ctype.h, wctype.h
	Category
	Classification Routines
	Prototype
	`int isascii(int c);`
	`int iswascii(wint_t c);`
	Description
	Character classification macro.
	These functions depend on the LC_CTYPE
	isascii is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false.
	isascii is defined on all integer values.
	Return Value
	isascii returns nonzero if c is in the range 0 to 127 (0x00-0x7F).
	iswascii returns nonzero if c is is a wide-character representation of an ASCII character.
	Each of these routines returns 0 if c does not satisfy the test condition.
	Example... more (see page 776)

iscntrl, iswcntrl (see page 777)	**Header File**
	ctype.h
	Category
	Classification Routines
	Prototype
	`int iscntrl(int c);`
	`int iswcntrl(wint_t c);`
	Description
	Tests for a control character.
	iscntrl is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a delete character or control character (0x7F or 0x00 to 0x1F).
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	iscntrl returns nonzero if c is a delete character or ordinary control character.
	Example

isdigit, iswdigit, _ismbcdigit (see page 778)	**Header File**
	ctype.h, mbstring.h
	Category
	Classification Routines
	Prototype
	`int isdigit(int c);`
	`int iswdigit(wint_t c);`
	`int _ismbcdigit(unsigned int c);`
	Description
	Tests for decimal-digit character.
	isdigit is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a digit (0 to 9).
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	isdigit returns nonzero if c is a digit.
	_ismbcdigit returns true if and only if the argument c is a single-byte representation of an ASCII digit.... more (see page 778)

isgraph, iswgraph, _ismbcgraph (see page 779)	**Header File**
	ctype.h, mbstring.h
	Category
	Classification Routines
	Prototype
	`int isgraph(int c);`
	`int iswgraph(wint_t c);`
	`int _ismbcgraph( unsigned int c);`
	Description
	Tests for printing character.
	isgraph is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a printing character except blank space (' ').
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	isgraph returns nonzero if c is a printing character.
	Example

islower, iswlower, _ismbclower (see page 779)	**Header File**
	ctype.h, mbstring.h
	Category
	Classification Routines
	Prototype
	`int islower(int c);`
	`int iswlower(wint_t c);`
	`int _ismbclower(unsigned int c);`
	Description
	Tests for lowercase character.
	islower is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a lowercase letter (a to z).
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	islower returns nonzero if c is a lowercase letter.
	Example

isprint, iswprint, _ismbcprint (see page 780)	**Header File**
	ctype.h, wctype.h, mbstring.h
	Category
	Classification Routines
	Prototype
	`int isprint(int c);`
	`int iswprint(wint_t c);`
	`int _ismbcprint(unsigned int c);`
	Description
	Tests for printing character.
	isprint is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a printing character including the blank space (' ').
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	isprint returns nonzero if c is a printing character.
	Example

ispunct, iswpunct, _ismbcpunct (see page 781)	**Header File**
	ctype.h, wctype.h, mbstring.h
	Category
	Classification Routines
	Prototype
	`int ispunct(int c);`
	`int iswpunct(wint_t c);`
	`int _ismbcpunct(unsigned int c);`
	Description
	Tests for punctuation character.
	ispunct is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is any printing character that is neither an alphanumeric nor a blank space (' ').
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	ispunct returns nonzero if c is a punctuation character.
	Example

isspace, iswspace, _ismbcspace (see page 782)	**Header File**
	ctype.h, wctype.h, mbstring.h
	Category
	Classification Routines
	Prototype
	`int isspace(int c);`
	`int iswspace(wint_t c);`
	`int _ismbcspace(unsigned int c);`
	Description
	Tests for space character.
	isspace is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	isspace returns nonzero if c is a space, tab, carriage return, new line, vertical tab, formfeed (0x09 to 0x0D, 0x20), or any other locale-defined space character.
	Example

isupper, iswupper, _ismbcupper (see page 783)	**Header File**
	ctype.h, wctype.h, mbstring.h
	Category
	Classification Routines
	Prototype
	`int isupper(int c);`
	`int iswupper(wint_t c);`
	`int _ismbcupper(unsigned int c);`
	Description
	Tests for uppercase character.
	isupper is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is an uppercase letter (A to Z).
	You can make this macro available as a function by undefining (**#undef**) it.
	Return Value
	isupper returns nonzero if c is an uppercase letter.
	Example

isxdigit, iswxdigit (see page 784)	**Header File**
	ctype.h, wctype.h
	Category
	Classification Routines
	Prototype
	`int isxdigit(int c);`
	`int iswxdigit(wint_t c);`
	Description
	Tests for hexadecimal character.
	isxdigit is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.
	You can make this macro available as a function by undefining (#undef) it.
	Return Value
	isxdigit returns nonzero if c is a hexadecimal digit (0 to 9, A to F, a to f) or any other hexadecimal digit defined by the locale.
	Example

toascii (see page 784)	**Header File** ctype.h **Category** Conversion Routines **Prototype** `int toascii(int c);` **Description** Translates characters to ASCII format. toascii is a macro that converts the integer c to ASCII by clearing all but the lower 7 bits; this gives a value in the range 0 to 127. **Return Value** toascii returns the converted value of c. **Portability**

tolower, _mbctolower, towlower (see page 785)	**Header File**
	ctype.h, mbstring.h
	Category
	Conversion Routines
	Prototype
	`int tolower(int ch);`
	`int towlower(wint_t ch); // Unicode version`
	`unsigned int _mbctolower(unsigned int c);`
	Description
	Translates characters to lowercase.
	tolower is a function that converts an integer ch (in the range EOF to 255) to its lowercase value (a to z; if it was uppercase, A to Z). All others are left unchanged.
	Return Value
	tolower returns the converted value of ch if it is uppercase; it returns all others unchanged.
	Example

toupper, _mbctoupper, towupper (see page 786)	**Header File**
	ctype.h, mbstring.h
	Category
	Conversion Routines
	Prototype
	`int toupper(int ch);`
	`int towupper(wint_t ch); // Unicode version`
	`unsigned int _mbctoupper(unsigned int c);`
	Description
	Translates characters to uppercase.
	toupper is a function that converts an integer ch (in the range EOF to 255) to its uppercase value (A to Z; if it was lowercase, a to z). All others are left unchanged.
	towupper is the Unicode version of toupper. It is available when Unicode is defined.
	Return Value
	toupper returns the converted value of ch if it is lowercase; it returns all others unchanged.
	Example

3.1.4.4.1 _ctype

Header File

ctype.h

Syntax

```
extern char _ctype[];
```

Description

_ctype is an array of character attribute information indexed by ASCII value + 1. Each entry is a set of bits describing the character. This array is used by isdigit, isprint, and so on.

3.1.4.4.2 _IS_xxx #defines

Header File

ctype.h

Description

Bit settings in the _ctype[] used by the is... character macros.

Name	Meaning
_IS_SP	Is space
_IS_DIG	Is digit
_IS_UPP	Is uppercase
_IS_LOW	Is lowercase
_IS_HEX	[A-F] or [a-f]
_IS_CTL	Control
_IS_PUN	Punctuation

3.1.4.4.3 _tolower

Header File

ctype.h

Category

Conversion Routines

Prototype

```
int _tolower(int ch);
```

Description

_tolower is a macro that does the same conversion as tolower, except that it should be used only when ch is known to be uppercase (AZ).

To use _tolower, you must include ctype.h.

Return Value

_tolower returns the converted value of ch if it is uppercase; otherwise, the result is undefined.

Example

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int length, i;
   char *string = "THIS IS A STRING.";
   length = strlen(string);
   for (i = 0; i < length; i++) {
       if ((string[i] >= 'A') && (string[i] <= 'Z')){
          string[i] = _tolower(string[i]);
       }
   }
   printf("%s\n",string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.4.4 _toupper

Header File

ctype.h

Category

Conversion Routines

Prototype

```
int _toupper(int ch);
```

Description

Translates characters to uppercase.

_toupper is a macro that does the same conversion as toupper, except that it should be used only when ch is known to be

lowercase (a to z).

To use _toupper, you must include ctype.h.

Return Value

_toupper returns the converted value of ch if it is lowercase; otherwise, the result is undefined.

Example

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int length, i;
   char *string = "this is a string.";
   length = strlen(string);
   for (i = 0; i < length; i++) {
      if ((string[i] >= 'a') && (string[i] <= 'z')){
         string[i] = _toupper(string[i]);
         }
   }
   printf("%s\n",string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.4.5 isalnum, __iscsym, iswalnum, _ismbcalnum

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isalnum(int c);
int __iscsym(int c);
int iswalnum(wint_t c);
int _ismbcalnum(unsigned int c);
```

Description

Tests for an alphanumeric character.

isalnum is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a letter (A to Z or a to z) or a digit (0 to 9).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

It is a predicate returning nonzero for true and 0 for false.

isalnum returns nonzero if c is a letter or a digit.

__iscsym returns nonzero if c is a letter, underscore, or digit.

iswalnum returns nonzero if iswalpha or iswdigit return true for c.

_ismbcalnum returns true if and only if the argument c is a single-byte ASCII English letter.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';

  if (isalnum(c))
    printf("%c is alphanumeric\n",c);
  else
    printf("%c is not alphanumeric\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isalnum	+	+	+	+
__iscsym		+		
_ismbcalnum		+		
iswalnum		+	+	+

3.1.4.4.6 isalpha, __iscsymf, iswalpha, _ismbcalpha

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isalpha(int c);
int __iscsymf(int c);
int iswalpha(wint_t c);
int _ismbcalpha(unsigned int c);
```

Description

Classifies an alphabetical character.

isalpha is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a letter (A to Z or a to z).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isalpha returns nonzero if c is a letter.

__iscsymf returns true if and only if the argument c is a letter or an underscore.

iswalpha returns nonzero if c is a **wchar_t** in the character set defined by the implementation.

_ismbcalpha returns true if and only if the argument c is a single-byte ASCII English letter.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';

  if (isalpha(c))
    printf("%c is alphabetical\n",c);
  else
    printf("%c is not alphabetical\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isalpha	+	+	+	+
__iscsymf		+		
_ismbcalpha		+		
iswalpha		+	+	+

3.1.4.4.7 isascii, iswascii

Header File

ctype.h, wctype.h

Category

Classification Routines

Prototype

```
int isascii(int c);
int iswascii(wint_t c);
```

Description

Character classification macro.

These functions depend on the LC_CTYPE

isascii is a macro that classifies ASCII-coded integer values by table lookup. It is a predicate returning nonzero for true and 0 for false.

isascii is defined on all integer values.

Return Value

isascii returns nonzero if c is in the range 0 to 127 (0x00-0x7F).

iswascii returns nonzero if c is is a wide-character representation of an ASCII character.

Each of these routines returns 0 if c does not satisfy the test condition.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <stdio.h>
int main(void)
{
  char c = 'C';
  if (isascii(c))
    printf("%c is ascii\n",c);
  else
    printf("%c is not ascii\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isascii		+		
iswascii		+		

3.1.4.4.8 iscntrl, iswcntrl

Header File

ctype.h

Category

Classification Routines

Prototype

```
int iscntrl(int c);
```

```
int iswcntrl(wint_t c);
```

Description

Tests for a control character.

iscntrl is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a delete character or control character (0x7F or 0x00 to 0x1F).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

iscntrl returns nonzero if c is a delete character or ordinary control character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (iscntrl(c))
    printf("%c is a control character\n",c);
  else
    printf("%c is not a control character\n",c);
  return 0;
```

```
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
iscntrl	+	+	+	+
iswcntrl		+	+	+

3.1.4.4.9 isdigit, iswdigit, _ismbcdigit

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isdigit(int c);
```

```
int iswdigit(wint_t c);
```

```
int _ismbcdigit(unsigned int c);
```

Description

Tests for decimal-digit character.

isdigit is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a digit (0 to 9).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isdigit returns nonzero if c is a digit.

_ismbcdigit returns true if and only if the argument c is a single-byte representation of an ASCII digit.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isdigit(c))
    printf("%c is a digit\n",c);
  else
    printf("%c is not a digit\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isdigit	+	+	+	+
_ismbcdigit		+		
iswdigit		+	+	+

3.1.4.4.10 isgraph, iswgraph, _ismbcgraph

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isgraph(int c);
```

```
int iswgraph(wint_t c);
```

```
int _ismbcgraph( unsigned int c);
```

Description

Tests for printing character.

isgraph is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a printing character except blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isgraph returns nonzero if c is a printing character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isgraph(c))
    printf("%c is a graphic character\n",c);
  else
    printf("%c is not a graphic character\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isgraph	+	+	+	+
_ismbcgraph		+		
iswgraph		+	+	+

3.1.4.4.11 islower, iswlower, _ismbclower

Header File

ctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int islower(int c);

int iswlower(wint_t c);

int _ismbclower(unsigned int c);
```

Description

Tests for lowercase character.

islower is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a lowercase letter (a to z).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

islower returns nonzero if c is a lowercase letter.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (islower(c))
    printf("%c is a lowercase character\n",c);
  else
    printf("%c is not a lowercase character\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
islower	+	+	+	+
_ismbclower		+		
iswlower		+	+	+

3.1.4.4.12 isprint, iswprint, _ismbcprint

Header File

ctype.h, wctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isprint(int c);

int iswprint(wint_t c);

int _ismbcprint(unsigned int c);
```

Description

Tests for printing character.

isprint is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is a printing character including the blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isprint returns nonzero if c is a printing character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isprint(c))
    printf("%c is a printable character\n",c);
  else
    printf("%c is not a printable character\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isprint	+	+	+	+
_ismbcprint		+		
iswprint		+		

3.1.4.4.13 ispunct, iswpunct, _ismbcpunct

Header File

ctype.h, wctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int ispunct(int c);
```

```
int iswpunct(wint_t c);
```

```
int _ismbcpunct(unsigned int c);
```

Description

Tests for punctuation character.

ispunct is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is any printing character that is neither an alphanumeric nor a blank space (' ').

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

ispunct returns nonzero if c is a punctuation character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (ispunct(c))
    printf("%c is a punctuation character\n",c);
  else
    printf("%c is not a punctuation character\n",c);

  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ispunct	+	+	+	+
_ismbcpunt		+		
iswpunct		+	+	+

3.1.4.4.14 isspace, iswspace, _ismbcspace

Header File

ctype.h, wctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isspace(int c);
```

```
int iswspace(wint_t c);
```

```
int _ismbcspace(unsigned int c);
```

Description

Tests for space character.

isspace is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isspace returns nonzero if c is a space, tab, carriage return, new line, vertical tab, formfeed (0x09 to 0x0D, 0x20), or any other locale-defined space character.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isspace(c))
    printf("%c is white space\n",c);
  else
    printf("%c is not white space\n",c);
```

```
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isspace	+	+	+	+
_ismbcspace		+		
iswspace		+	+	+

3.1.4.4.15 isupper, iswupper, _ismbcupper

Header File

ctype.h, wctype.h, mbstring.h

Category

Classification Routines

Prototype

```
int isupper(int c);
```

```
int iswupper(wint_t c);
```

```
int _ismbcupper(unsigned int c);
```

Description

Tests for uppercase character.

isupper is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category. For the default C locale, c is an uppercase letter (A to Z).

You can make this macro available as a function by undefining (**#undef**) it.

Return Value

isupper returns nonzero if c is an uppercase letter.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isupper(c))
    printf("%c is an uppercase character\n",c);
  else
    printf("%c is not an uppercase character\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isupper	+	+	+	+
_ismbcupper		+		
iswupper		+	+	+

3.1.4.4.16 isxdigit, iswxdigit

Header File

ctype.h, wctype.h

Category

Classification Routines

Prototype

```
int isxdigit(int c);
```

```
int iswxdigit(wint_t c);
```

Description

Tests for hexadecimal character.

isxdigit is a macro that classifies ASCII-coded integer values by table lookup. The macro is affected by the current locale's LC_CTYPE category.

You can make this macro available as a function by undefining (#undef) it.

Return Value

isxdigit returns nonzero if c is a hexadecimal digit (0 to 9, A to F, a to f) or any other hexadecimal digit defined by the locale.

Example

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
  char c = 'C';
  if (isxdigit(c))
    printf("%c is a hexadecimal digit\n",c);
  else
    printf("%c is not a hexadecimal digit\n",c);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
isxdigit	+	+	+	+
iswxdigit		+	+	+

3.1.4.4.17 toascii

Header File

ctype.h

Category

Conversion Routines

Prototype

```
int toascii(int c);
```

Description

Translates characters to ASCII format.

toascii is a macro that converts the integer c to ASCII by clearing all but the lower 7 bits; this gives a value in the range 0 to 127.

Return Value

toascii returns the converted value of c.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.4.18 tolower, _mbctolower, towlower

Header File

ctype.h, mbstring.h

Category

Conversion Routines

Prototype

```
int tolower(int ch);
```

```
int towlower(wint_t ch); // Unicode version
```

```
unsigned int _mbctolower(unsigned int c);
```

Description

Translates characters to lowercase.

tolower is a function that converts an integer ch (in the range EOF to 255) to its lowercase value (a to z; if it was uppercase, A to Z). All others are left unchanged.

Return Value

tolower returns the converted value of ch if it is uppercase; it returns all others unchanged.

Example

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int length, i;
   char *string = "THIS IS A STRING";
   length = strlen(string);
   for (i=0; i<length; i++)
   {
       string[i] = tolower(string[i]);
   }
   printf("%s\n",string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
tolower	+	+	+	+
_mbctolower		+		
towlower		+	+	+

3.1.4.4.19 toupper, _mbctoupper, towupper

Header File

ctype.h, mbstring.h

Category

Conversion Routines

Prototype

```
int toupper(int ch);
int towupper(wint_t ch); // Unicode version
unsigned int _mbctoupper(unsigned int c);
```

Description

Translates characters to uppercase.

toupper is a function that converts an integer ch (in the range EOF to 255) to its uppercase value (A to Z; if it was lowercase, a to z). All others are left unchanged.

towupper is the Unicode version of toupper. It is available when Unicode is defined.

Return Value

toupper returns the converted value of ch if it is lowercase; it returns all others unchanged.

Example

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int length, i;
   char *string = "this is a string";
   length = strlen(string);
   for (i=0; i<length; i++)
   {
      string[i] = toupper(string[i]);
   }
   printf("%s\n",string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
toupper	+	+	+	+
_mbctoupper		+		
towupper		+	+	+

3.1.4.5 delayimp.h

The following functions, macros, and classes are provided in `delayimp.h`:

Topics

Name	Description

__pfnDliNotifyHook, __pfnDliFailureHook (see page 787)	**Header File**
	delayimp.h
	Category
	Delay load hook notification Routines
	Prototype
	`typedef FARPROC (WINAPI *DelayedLoadHook)(dliNotification dliNotify,`
	`DelayLoadInfo * pdli);`
	Description
	The delay load mechanism provides two hooks for you to modify the runtime behavior of a delay loaded DLL. By writing your own hook functions using the function signature below and assigning this to the two hooks you can modify the delay load process.
	DelayLoadProc structure
	`typedef struct DelayLoadProc`
	`{`
	`BOOL fImportByName;`
	`union`
	`{`
	`LPCSTR szProcName;`
	`DWORD dwOrdinal;`
	`};`
	`} DelayLoadProc;`
	ImgDelayDescr structure
	`typedef struct ImgDelayDescr`
	`{`
	`DWORD grAttrs; /* attributes */`
	`LPCSTR szName; /* pointer to dll name */`
	`HMODULE...  more (see page 787)`

__FUnloadDelayLoadedDLL (see page 790)	**Header File** delayimp.h **Prototype** BOOL WINAPI __FUnloadDelayLoadedDLL(LPCSTR szDll); **Description** Unloads a delay loaded DLL. szDll is pointer to a name to unload, or NULL to unload all the delay load DLLs in the list. **Return Value** On successful completion __FUnloadDelayLoadedDLL returns **true**. On error it returns **false**.

3.1.4.5.1 __pfnDliNotifyHook, __pfnDliFailureHook

Header File

delayimp.h

Category

Delay load hook notification Routines

Prototype

```
typedef FARPROC (WINAPI *DelayedLoadHook)(dliNotification dliNotify,
DelayLoadInfo * pdli);
```

Description

The delay load mechanism provides two hooks for you to modify the runtime behavior of a delay loaded DLL. By writing your own hook functions using the function signature below and assigning this to the two hooks you can modify the delay load process.

DelayLoadProc structure

```
typedef struct DelayLoadProc
{
BOOL fImportByName;
union
{
LPCSTR szProcName;
DWORD dwOrdinal;
};
} DelayLoadProc;
```

ImgDelayDescr structure

```
typedef struct ImgDelayDescr
{
DWORD grAttrs; /* attributes */
LPCSTR szName; /* pointer to dll name */
HMODULE hmod; /* address of module handle */
IMAGE_THUNK_DATA * pIAT; /* address of the IAT */
IMAGE_THUNK_DATA * pINT; /* address of the INT */
IMAGE_THUNK_DATA * pBoundIAT; /* address of the optional bound IAT */
IMAGE_THUNK_DATA * pUnloadIAT; /* address of optional copy of
original IAT */
DWORD dwTimeStamp; /* 0 if not bound, */
/* O.W. date/time stamp of DLL bound
to (Old BIND) */
} ImgDelayDescr;
```

DelayLoadInfo structure

```
typedef struct DelayLoadInfo
{
DWORD cb; /* size of structure */
const ImgDelayDescr * pidd; /* raw form of data (everything is
there) */
FARPROC * ppfn; /* points to address of function to
load */
```

```
LPCSTR szDll; /* name of dll */
DelayLoadProc dlp; /* name or ordinal of procedure */
HMODULE hmodCur; /* the hInstance of the library we
have loaded */
FARPROC pfnCur; /* the actual function that will be
called */
DWORD dwLastError;/* error received (if an error
notification) */
} DelayLoadInfo, *PDelayLoadInfo;
```

Delay load import hook notifications

The following structure is the enumerations that are defined for the hook notification events:

```
typedef enum
{
dliNoteStartProcessing, /* used to bypass or note helper only */
dliNotePreLoadLibrary, /* called just before LoadLibrary, can */
/* override w/ new HMODULE return val */
dliNotePreGetProcAddress, /* called just before GetProcAddress, can */
/* override w/ new FARPROC return value */
dliFailLoadLibrary, /* failed to load library, fix it by */
/* returning a valid HMODULE */
dliFailGetProcAddress, /* failed to get proc address, fix it by */
/* returning a valid FARPROC */
dliNoteEndProcessing, /* called after all processing is done, */
/* no bypass possible at this point */
/* except by longjmp(), throw(), or
RaiseException. */
} dliNotification;
```

Hook pointers

The “notify hook” gets called for every call to the delay load helper. This allows a user to hook every call and skip the delay load helper entirely.

```
extern DelayedLoadHook _EXPDATA __pfnDliNotifyHook;
dliNotify ==
{
dliNoteStartProcessing |
dliNotePreLoadLibrary |
dliNotePreGetProcAddress |
dliNoteEndProcessing
```

```
}
```

Note: The “failure” hook is assigned to:

```
extern DelayedLoadHook _EXPDATA __pfnDliFailureHook;
```

This hook is called with the following notification flags:

```
dliNotify ==
{
dliFailLoadLibrary |
dliFailGetProcAddress
}
```

For further information on when this notify eventsd occur during the delay load process, please see delayhlp.c

3.1.4.5.2 __FUnloadDelayLoadedDLL

Header File

delayimp.h

Prototype

```
BOOL WINAPI __FUnloadDelayLoadedDLL(LPCSTR szDll);
```

Description

Unloads a delay loaded DLL.

szDll is pointer to a name to unload, or NULL to unload all the delay load DLLs in the list.

Return Value

On successful completion __FUnloadDelayLoadedDLL returns **true**.

On error it returns **false**.

3.1.4.6 direct.h

The following functions, macros, and classes are provided in `direct.h`:

Topics

Name	Description
_chdrive (see page 791)	**Header File** direct.h **Category** Directory Control Routines **Prototype** `int _chdrive(int drive);` **Description** Sets current disk drive. _chdrive sets the current drive to the one associated with drive: 1 for A, 2 for B, 3 for C, and so on. This function changes the current drive of the parent process. **Return Value** _chdrive returns 0 if the current drive was changed successfully; otherwise, it returns -1. **Example**

_getdcwd, _wgetdcwd (see page 791)	**Header File**
	direct.h
	Category
	Directory Control Routines
	Prototype
	`char * _getdcwd(int drive, char *buffer, int buflen);`
	`wchar_t * _wgetdcwd(int drive, wchar_t *buffer, int buflen);`
	Description
	Gets current directory for specified drive.
	_getdcwd gets the full path name of the working directory of the specified drive (including the drive name), up to buflen bytes long, and stores it in buffer. If the full path name length (including the null-terminator) is longer than buflen, an error occurs. The drive is 0 for the default drive, 1=A, 2=B, and so on.
	If the working directory is the root directory, the terminating character for... more (see page 791)

3.1.4.6.1 _chdrive

Header File

direct.h

Category

Directory Control Routines

Prototype

```
int _chdrive(int drive);
```

Description

Sets current disk drive.

_chdrive sets the current drive to the one associated with drive: 1 for A, 2 for B, 3 for C, and so on.

This function changes the current drive of the parent process.

Return Value

_chdrive returns 0 if the current drive was changed successfully; otherwise, it returns -1.

Example

```
 #include <stdio.h>
#include <direct.h>
int main(void)
{
  if (_chdrive(3) == 0)
    printf("Successfully changed to drive C:\n");
  else
    printf("Cannot change to drive C:\n");
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.6.2 _getdcwd, _wgetdcwd

Header File

direct.h

Category

Directory Control Routines

Prototype

```
char * _getdcwd(int drive, char *buffer, int buflen);
```

```
wchar_t * _wgetdcwd(int drive, wchar_t *buffer, int buflen);
```

Description

Gets current directory for specified drive.

_getdcwd gets the full path name of the working directory of the specified drive (including the drive name), up to buflen bytes long, and stores it in buffer. If the full path name length (including the null-terminator) is longer than buflen, an error occurs. The drive is 0 for the default drive, 1=A, 2=B, and so on.

If the working directory is the root directory, the terminating character for the full path is a backslash. If the working directory is a subdirectory, there is no terminating backslash after the subdirectory name.

If buffer is NULL, _getdcwd allocates a buffer at least buflen bytes long. You can later free the allocated buffer by passing the _getdcwd return value to the free function.

Return Value

If successful, _getdcwd returns a pointer to the buffer containing the current directory for the specified drive.

Otherwise it returns NULL, and sets the global variable errno to one of the following values:

ENOMEM	Not enough memory to allocate a buffer (buffer is NULL)
ERANGE	Directory name longer than buflen (buffer is not NULL)

Example

```
#include <direct.h>
#include <stdio.h>
char buf[65];
void main()
{
  if (_getdcwd(3, buf, sizeof(buf)) == NULL)
    perror("Unable to get current directory of drive C");
  else
    printf("Current directory of drive C is %s\n",buf);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_getdcwd		+		
_wgetdcwd		NT only		

3.1.4.7 dirent.h

The following functions, macros, and classes are provided in `dirent.h`:

Topics

Name	Description

closedir, wclosedir (see page 794)	**Header File**
	dirent.h
	Category
	Directory Control Routines
	Prototype
	`void closedir(DIR *dirp);`
	`void wclosedir(wDIR *dirp);`
	Description
	Closes a directory stream.
	closedir is available on POSIX-compliant UNIX systems.
	The closedir function closes the directory stream dirp, which must have been opened by a previous call to opendir. After the stream is closed, dirp no longer points to a valid directory stream.
	wclosedir is the Unicode version of closedir.
	Return Value
	If closedir is successful, it returns 0. Otherwise, closedir returns -1 and sets the global variable errno to

opendir, wopendir (see page 795)	**Header File**
	dirent.h
	Category
	Directory Control Routines
	Prototype
	`DIR *opendir(const char *dirname);`
	`wDIR *wopendir(const wchar_t *dirname);`
	Description
	Opens a directory stream for reading.
	opendir is available on POSIX-compliant UNIX systems.
	The opendir function opens a directory stream for reading. The name of the directory to read is dirname. The stream is set to read the first entry in the directory.
	A directory stream is represented by the DIR structure, defined in dirent.h. This structure contains no user-accessible fields. Multiple directory streams can be opened and read simultaneously. Directory entries can be created or deleted while a directory stream is being... more (see page 795)

readdir, wreaddir (see page 796)	**Header File**
	dirent.h
	Category
	Directory Control Routines
	Prototype
	`struct dirent *readdir(DIR *dirp);`
	`struct wdirent *wreaddir(wDIR *dirp)`
	Description
	Reads the current entry from a directory stream.
	readdir is available on POSIX-compliant UNIX systems.
	The readdir function reads the current directory entry in the directory stream pointed to by dirp. The directory stream is advanced to the next entry.
	The readdir function returns a pointer to a **dirent** structure that is overwritten by each call to the function on the same directory stream. The structure is not overwritten by a readdir call on a different directory stream.
	The **dirent** structure corresponds to... more (see page 796)

rewinddir, wrewinddir (see page 797)	**Header File**
	dirent.h
	Category
	Directory Control Routines
	Prototype
	`void rewinddir(DIR *dirp);`
	`void wrewinddir(wDIR *dirp);`
	Description
	Resets a directory stream to the first entry.
	rewinddir is available on POSIX-compliant UNIX systems.
	The rewinddir function repositions the directory stream dirp at the first entry in the directory. It also ensures that the directory stream accurately reflects any directory entries that might have been created or deleted since the last opendir or rewinddir on that directory stream.
	wrewinddir is the Unicode version of rewinddir.
	Return Value
	None.
	Example

3.1.4.7.1 closedir, wclosedir

Header File

dirent.h

Category

Directory Control Routines

Prototype

```
void closedir(DIR *dirp);
```

```
void wclosedir(wDIR *dirp);
```

Description

Closes a directory stream.

closedir is available on POSIX-compliant UNIX systems.

The closedir function closes the directory stream dirp, which must have been opened by a previous call to opendir. After the stream is closed, dirp no longer points to a valid directory stream.

wclosedir is the Unicode version of closedir.

Return Value

If closedir is successful, it returns 0. Otherwise, closedir returns -1 and sets the global variable errno to

EBADF	The dirp argument does not point to a valid open directory stream

Example

```
/* opendir.c - test opendir(), readdir(), closedir() */

#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

void scandir(char *dirname)
{
   DIR *dir;
   struct dirent *ent;
```

```
    printf("First pass on '%s':\n",dirname);
    if ((dir = opendir(dirname)) == NULL)
    {
      perror("Unable to open directory");
      exit(1);
    }
    while ((ent = readdir(dir)) != NULL)
      printf("%s\n",ent->d_name);

    printf("Second pass on '%s':\n",dirname);
    rewinddir(dir);
    while ((ent = readdir(dir)) != NULL)
      printf("%s\n",ent->d_name);
    if (closedir(dir) != 0)
      perror("Unable to close directory");
}

void main(int argc,char *argv[])
{
    if (argc != 2)
    {
      printf("usage: opendir dirname\n");
      exit(1);
    }
    scandir(argv[1]);
    exit(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.7.2 opendir, wopendir

Header File

dirent.h

Category

Directory Control Routines

Prototype

```
DIR *opendir(const char *dirname);
```

```
wDIR *wopendir(const wchar_t *dirname);
```

Description

Opens a directory stream for reading.

opendir is available on POSIX-compliant UNIX systems.

The opendir function opens a directory stream for reading. The name of the directory to read is dirname. The stream is set to read the first entry in the directory.

A directory stream is represented by the DIR structure, defined in dirent.h. This structure contains no user-accessible fields. Multiple directory streams can be opened and read simultaneously. Directory entries can be created or deleted while a directory stream is being read.

Use the readdir function to read successive entries from a directory stream. Use the closedir function to remove a directory

stream when it is no longer needed.

Return Value

On success, opendir returns a pointer to a directory stream that can be used in calls to readdir, rewinddir, and closedir.

On error (If the directory cannot be opened), the functino returns NULL and sets the global variable errno to

ENOENT	The directory does not exist
ENOMEM	Not enough memory to allocate a DIR object

Example

```
 /* opendir.c - test opendir(), readdir(), closedir() */

#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

void scandir(char *dirname)
{
   DIR *dir;
   struct dirent *ent;

   printf("First pass on '%s':\n",dirname);
   if ((dir = opendir(dirname)) == NULL)
   {
     perror("Unable to open directory");
     exit(1);
   }
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);

   printf("Second pass on '%s':\n",dirname);
   rewinddir(dir);
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);
   if (closedir(dir) != 0)
     perror("Unable to close directory");
}

void main(int argc,char *argv[])
{
   if (argc != 2)
   {
     printf("usage: opendir dirname\n");
     exit(1);
   }
   scandir(argv[1]);
   exit(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
opendir	+	+		
wopendir				

3.1.4.7.3 readdir, wreaddir

Header File

dirent.h

Category

Directory Control Routines

Prototype

```
struct dirent *readdir(DIR *dirp);
struct wdirent *wreaddir(wDIR *dirp)
```

Description

Reads the current entry from a directory stream.

readdir is available on POSIX-compliant UNIX systems.

The readdir function reads the current directory entry in the directory stream pointed to by dirp. The directory stream is advanced to the next entry.

The readdir function returns a pointer to a **dirent** structure that is overwritten by each call to the function on the same directory stream. The structure is not overwritten by a readdir call on a different directory stream.

The **dirent** structure corresponds to a single directory entry. It is defined in dirent.h and contains (in addition to other non-accessible members) the following member:

char d_name[];

where d_name is an array of characters containing the null-terminated file name for the current directory entry. The size of the array is indeterminate; use strlen to determine the length of the file name.

All valid directory entries are returned, including subdirectories, “.” and “..” entries, system files, hidden files, and volume labels. Unused or deleted directory entries are skipped.

A directory entry can be created or deleted while a directory stream is being read, but readdir might or might not return the affected directory entry. Rewinding the directory with rewinddir or reopening it with opendir ensures that readdir will reflect the current state of the directory.

The wreaddir function is the Unicode version of readdir. It uses the **wdirent** structure but otherwise is similar to readdir.

Return Value

On success, readdir returns a pointer to the current directory entry for the directory stream.

If the end of the directory has been reached, or dirp does not refer to an open directory stream, readdir returns NULL.

Portability

	POSIX	Win32	ANSI C	ANSI C++
readdir	+	+		
wreaddir		+		

3.1.4.7.4 rewinddir, wrewinddir

Header File

dirent.h

Category

Directory Control Routines

Prototype

```
void rewinddir(DIR *dirp);

void wrewinddir(wDIR *dirp);
```

Description

Resets a directory stream to the first entry.

rewinddir is available on POSIX-compliant UNIX systems.

The rewinddir function repositions the directory stream dirp at the first entry in the directory. It also ensures that the directory stream accurately reflects any directory entries that might have been created or deleted since the last opendir or rewinddir on that directory stream.

wrewinddir is the Unicode version of rewinddir.

Return Value

None.

Example

```
/* opendir.c - test opendir(), readdir(), closedir() */

#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>

void scandir(char *dirname)
{
   DIR *dir;
   struct dirent *ent;

   printf("First pass on '%s':\n",dirname);
   if ((dir = opendir(dirname)) == NULL)
   {
     perror("Unable to open directory");
     exit(1);
   }
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);

   printf("Second pass on '%s':\n",dirname);
   rewinddir(dir);
   while ((ent = readdir(dir)) != NULL)
     printf("%s\n",ent->d_name);
   if (closedir(dir) != 0)
     perror("Unable to close directory");
 }

 void main(int argc,char *argv[])
 {
   if (argc != 2)
   {
     printf("usage: opendir dirname\n");
     exit(1);
   }
   scandir(argv[1]);
   exit(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
rewind	+	+		
wrewinddir		+		

3.1.4.8 dir.h

The following functions, macros, and classes are provided in `dir.h`:

Topics

Name	Description

chdir (see page 803)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`int chdir(const char *path);`
	`int _wchdir(const wchar_t *path);`
	Description
	Changes current directory.
	chdir causes the directory specified by path to become the current working directory; path must specify an existing directory.
	A drive can also be specified in the path argument, such as
	`chdir("a:\\BC")`
	but this method changes only the current directory on that drive; it does not change the active drive.
	• Under Windows, only the current process is affected.
	Return Value
	Upon successful completion, chdir returns a value of 0. Otherwise, it returns a value of -1, and the global variable... more (see page 803)

MAXxxxx #defines (fnsplit) (see page 804)	**Header File** dir.h **Description** These symbols define the maximum number of characters in a file specification for fnsplit (including room for a terminating NULL).

findclose, _wfindclose (see page 804)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`int findclose(struct ffblk *ffblk );`
	`int _wfindclose(struct _wffblk *ffblk );`
	Description
	findclose closes any handles and frees up any dynamic memory associated with previous calls to findfirst and findnext.
	Return Value
	findclose returns 0 on successfully closing the handle. On failure:
	• -1 is returned
	• errno is set to

findfirst, _wfindfirst (see page 805)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);`
	`int _wfindfirst( const wchar_t *pathname, struct _wffblk *ffblk, int attrib);`
	Description
	Searches a disk directory.
	findfirst begins a search of a disk directory for files specified by attributes or wildcards.
	pathname is a string with an optional drive specifier path and file name of the file to be found. Only the file name portion can contain wildcard match characters (such as ? or *). If a matching file is found the ffblk structure is filled with the file-directory information.
	When Unicode is defined,... more (see page 805)

findnext, _wfindnext (see page 807)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`int findnext(struct ffblk *ffblk );`
	`int _wfindnext(struct _wffblk *ffblk );`
	Description
	Continues findfirst search.
	findnext is used to fetch subsequent files that match the pathname given in findfirst. ffblk is the same block filled in by the findfirst call. This block contains necessary information for continuing the search. One file name for each call to findnext will be returned until no more files are found in the directory matching the pathname.
	Return Value
	findnext returns 0 on successfully finding a file matching the search pathname. When no more files can be found... more (see page 807)

fnmerge, _wfnmerge (see page 808)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`void fnmerge(char *path, const char *drive, const char *dir, const char *name, const char *ext);`
	`void _wfnmerge(wchar_t *path, const wchar_t *drive, const wchar_t *dir, const wchar_t *name, const wchar_t *ext );`
	Description
	Builds a path from component parts.
	fnmerge makes a path name from its components. The new path name is
	`X:\DIR\SUBDIR\NAME.EXT`
	where:

fnsplit, _wfnsplit (see page 810)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`int fnsplit(const char *path, char *drive, char *dir, char *name, char *ext);`
	`int _wfnsplit(const wchar_t *path, wchar_t *drive, wchar_t *dir, wchar_t *name, wchar_t *ext );`
	Description
	Splits a full path name into its components.
	fnsplit takes a file's full path name (path) as a string in the form X:\DIR\SUBDIR\NAME.EXT and splits path into its four components. It then stores those components in the strings pointed to by drive, dir, name, and ext. All five components must be passed but any of them can be a null which means the corresponding component will... more (see page 810)

Bit Definitions for fnsplit (see page 811)	**Header File** dir.h **Description** Bit definitions returned from fnsplit to identify which pieces of a file name were found during the split.

getcurdir, _wgetcurdir (see page 811)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`int getcurdir(int drive, char *directory);`
	`int _wgetcurdir(int drive, wchar_t *directory );`
	Description
	Gets current directory for specified drive.
	getcurdir gets the name of the current working directory for the drive indicated by drive. drive specifies a drive number (0 for default, 1 for A, and so on). directory points to an area of memory of length MAXDIR where the null-terminated directory name will be placed. The name does not contain the drive specification and does not begin with a backslash.
	Return Value
	getcurdir returns 0 on success or -1 in the event... more (see page 811)

getcwd, _wgetcwd (see page 812)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`char *getcwd(char *buf, int buflen);`
	`wchar_t *_wgetcwd(wchar_t *buf, int buflen);`
	Description
	Gets current working directory.
	getcwd gets the full path name (including the drive) of the current working directory, up to buflen bytes long and stores it in buf. If the full path name length (including the null terminator) is longer than buflen bytes, an error occurs.
	If buf is NULL, a buffer buflen bytes long is allocated for you with malloc. You can later free the allocated buffer by passing the return value of getcwd to the function free.
	Return Value... more (see page 812)

getdisk, setdisk (see page 813)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`int getdisk(void);`
	`int setdisk(int drive);`
	Description
	Gets or sets the current drive number.
	getdisk gets the current drive number. It returns an integer: 0 for A, 1 for B, 2 for C, and so on.
	setdisk sets the current drive to the one associated with drive: 0 for A, 1 for B, 2 for C, and so on.
	The setdisk function changes the current drive of the parent process.
	Return Value
	getdisk returns the current drive number. setdisk returns the total number of drives available.
	Example

mkdir, _wmkdir (see page 814)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`int mkdir(const char *path);`
	`int _wmkdir(const wchar_t *path);`
	Description
	Creates a directory.
	mkdir is available on UNIX, though it then takes an additional parameter.
	mkdir creates a new directory from the given path name path.
	Return Value
	mkdir returns the value 0 if the new directory was created.
	A return value of -1 indicates an error, and the global variable errno is set to one of the following values:

_mktemp, _wmktemp (see page 815)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`char *_mktemp(char *template);`
	`wchar_t *_wmktemp(wchar_t *template);`
	Description
	Makes a unique file name.
	_mktemp replaces the string pointed to by template with a unique file name and returns template.
	template should be a null-terminated string with six trailing Xs. These Xs are replaced with a unique collection of letters plus a period, so that there are two letters, a period, and three suffix letters in the new file name.
	Starting with AA.AAA, the new file name is assigned by looking up the name on the disk and avoiding pre-existing names of the same... more (see page 815)

_rmdir, _wrmdir (see page 816)	**Header File**
	dir.h
	Category
	Directory Control Routines
	Prototype
	`int _rmdir(const char *path);`
	`int _wrmdir(const wchar_t *path);`
	Description
	Removes a directory.
	_rmdir deletes the directory whose path is given by path. The directory named by path
	• must be empty
	• must not be the current working directory
	• must not be the root directory
	Return Value
	_rmdir returns 0 if the directory is successfully deleted. A return value of -1 indicates an error, and the global variable errno is set to one of the following values:

searchpath, wsearchpath (see page 817)	**Header File**
	dir.h
	Category
	Miscellaneous Routines
	Prototype
	`char *searchpath(const char *file);`
	`wchar_t *wsearchpath( const wchar_t *file );`
	Description
	Searches the operating system path for a file.
	searchpath attempts to locate file, searching along the operating system path, which is the PATH=... string in the environment. A pointer to the complete path-name string is returned as the function value.
	searchpath searches for the file in the current directory of the current drive first. If the file is not found there, the PATH environment variable is fetched, and each directory in the path is searched in turn until the file is found,... more (see page 817)

3.1.4.8.1 chdir

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int chdir(const char *path);
```

```
int _wchdir(const wchar_t *path);
```

Description

Changes current directory.

chdir causes the directory specified by path to become the current working directory; path must specify an existing directory.

A drive can also be specified in the path argument, such as

```
chdir("a:\\BC")
```

but this method changes only the current directory on that drive; it does not change the active drive.

- Under Windows, only the current process is affected.

Return Value

Upon successful completion, chdir returns a value of 0. Otherwise, it returns a value of -1, and the global variable errno is set to

ENOENT	Path or file name not found

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <dir.h>
char old_dir[MAXDIR];
char new_dir[MAXDIR];
int main(void)
{
   if (getcurdir(0, old_dir))
   {
      perror("getcurdir()");
      exit(1);
```

```
    }
    printf("Current directory is: \\%s\n", old_dir);
    if (chdir("\\"))
    {
       perror("chdir()");
       exit(1);
    }
    if (getcurdir(0, new_dir))
    {
       perror("getcurdir()");
       exit(1);
    }
    printf("Current directory is now: \\%s\n", new_dir);
    printf("\nChanging back to original directory: \\%s\n", old_dir);
    if (chdir(old_dir))
    {
       perror("chdir()");
       exit(1);
    }
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
chdir	+	+		
_wchdir		NT only		

3.1.4.8.2 MAXxxxx #defines (fnsplit)

Header File

dir.h

Description

These symbols define the maximum number of characters in a file specification for fnsplit (including room for a terminating NULL).

Name	Meaning
MAXPATH	Complete file name with path
MAXDRIVE	Disk drive (e.g., "A:")
MAXDIR	File subdirectory specification
MAXFILE	File name without extension
MAXEXT	File extension

3.1.4.8.3 findclose, _wfindclose

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int findclose(struct ffblk *ffblk );
int _wfindclose(struct _wffblk *ffblk );
```

Description

findclose closes any handles and frees up any dynamic memory associated with previous calls to findfirst and findnext.

Return Value

findclose returns 0 on successfully closing the handle. On failure:

- -1 is returned
- errno is set to

EINVDAT	Invalid data

Portability

	POSIX	Win32	ANSI C	ANSI C++
findclose		+		
_wfindclose		NT only		

See Also

findfirst (see page 805)

findnext (see page 807)

3.1.4.8.4 findfirst, _wfindfirst

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);
int _wfindfirst( const wchar_t *pathname, struct _wffblk *ffblk, int attrib);
```

Description

Searches a disk directory.

findfirst begins a search of a disk directory for files specified by attributes or wildcards.

pathname is a string with an optional drive specifier path and file name of the file to be found. Only the file name portion can contain wildcard match characters (such as ? or *). If a matching file is found the ffblk structure is filled with the file-directory information.

When Unicode is defined, the _wfindfirst function uses the following _wffblk structure.

```
struct _wffblk {
```

```
long ff_reserved;
long ff_fsize;
unsigned long ff_attrib;
unsigned short ff_ftime;
unsigned short ff_fdate;
wchar_t ff_name[256];
};
```

For Win32, the format of the structure ffblk is as follows:

```
struct ffblk {
long ff_reserved;
long ff_fsize; /* file size */
unsigned long ff_attrib; /* attribute found */
unsigned short ff_ftime; /* file time */
unsigned short ff_fdate; /* file date */
char ff_name[256]; /* found file name */
};
```

attrib is a file-attribute byte used in selecting eligible files for the search. attrib should be selected from the following constants defined in dos.h:

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory
FA_ARCH	Archive

A combination of constants can be OR'ed together.

For more detailed information about these attributes refer to your operating system documentation.

ff_ftime and ff_fdate contain bit fields for referring to the current date and time. The structure of these fields was established by the operating system. Both are 16-bit structures divided into three fields.

ff_ftime:

Bits 0 to 4	The result of seconds divided by 2 (for example 10 here means 20 seconds)
Bits 5 to 10	Minutes
Bits 11 to 15	Hours

ff_fdate:

Bits 0-4	Day
Bits 5-8	Month
Bits 9-15	Years since 1980 (for example 9 here means 1989)

The structure ftime declared in io.h uses time and date bit fields similar in structure to ff_ftime and ff_fdate.

Return Value

findfirst returns 0 on successfully finding a file matching the search pathname.

When no more files can be found, or if there is an error in the file name:

- -1 is returned
- errno is set to

ENOENT	Path or file name not found

- _doserrno is set to one of the following values:

ENMFILE	No more files
ENOENT	Path or file name not found

Example

```
/* findfirst and findnext example */
#include <stdio.h>
#include <dir.h>
int main(void)
{
   struct ffblk ffblk;
   int done;
   printf("Directory listing of *.*\n");
   done = findfirst("*.*",&ffblk,0);
   while (!done)
   {
      printf("  %s\n", ffblk.ff_name);
      done = findnext(&ffblk);
   }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
findfirst		+		
_wfindfirst		NT only		

See Also

findclose (see page 804)

findnext (see page 807)

3.1.4.8.5 findnext, _wfindnext

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int findnext(struct ffblk *ffblk );
```

```
int _wfindnext(struct _wffblk *ffblk );
```

Description

Continues findfirst search.

findnext is used to fetch subsequent files that match the pathname given in findfirst. ffblk is the same block filled in by the findfirst call. This block contains necessary information for continuing the search. One file name for each call to findnext will be returned until no more files are found in the directory matching the pathname.

Return Value

findnext returns 0 on successfully finding a file matching the search pathname. When no more files can be found or if there is an error in the file name

-1 is returned

errno is set to

ENOENT	Path or file name not found

_doserrno is set to one of the following values:

ENMFILE	No more files
ENOENT	Path or file name not found

Portability

	POSIX	Win32	ANSI C	ANSI C++
findnext		+		
_wfindnext		NT only		

See Also

findfirst (see page 805)

3.1.4.8.6 fnmerge, _wfnmerge

Header File

dir.h

Category

Directory Control Routines

Prototype

```
void fnmerge(char *path, const char *drive, const char *dir, const char *name, const char *ext);
```

```
void _wfnmerge(wchar_t *path, const wchar_t *drive, const wchar_t *dir, const wchar_t *name, const wchar_t *ext );
```

Description

Builds a path from component parts.

fnmerge makes a path name from its components. The new path name is

`X:\DIR\SUBDIR\NAME.EXT`

where:

drive	=	X
dir	=	\\DIR\\SUBDIR\\
name	=	NAME
ext	=	.EXT

If drive is empty or NULL, no drive is inserted in the path name. If it is missing a trailing colon (:), a colon is inserted in the path name.

If dir is empty or NULL, no directory is inserted in the path name. If it is missing a trailing slash (\ or /), a backslash is inserted in the path name.

If name is empty or NULL, no file name is inserted in the path name.

If ext is empty or NULL, no extension is inserted in the path name. If it is missing a leading period (.), a period is inserted in the path name.

fnmerge assumes there is enough space in path for the constructed path name. The maximum constructed length is MAXPATH. MAXPATH is defined in dir.h.

fnmerge and fnsplit are invertible; if you split a given path with fnsplit then merge the resultant components with fnmerge you end up with path.

Return Value

None.

Example

```
#include <string.h>
#include <stdio.h>
#include <dir.h>
int main(void)
{
   char s[MAXPATH];
   char drive[MAXDRIVE];
   char dir[MAXDIR];
   char file[MAXFILE];
   char ext[MAXEXT];
   getcwd(s,MAXPATH);              /* get the current working directory */
   strcat(s,"\\");                 /* append on a trailing character */
   fnsplit(s,drive,dir,file,ext);  /* split the string to separate elems */
   strcpy(file,"DATA");
   strcpy(ext,".TXT");
   fnmerge(s,drive,dir,file,ext);   /* merge everything into one string */
   puts(s);                         /* display resulting string */
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fnmerge		+		
_wfnmerge		NT only		

3.1.4.8.7 fnsplit, _wfnsplit

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int fnsplit(const char *path, char *drive, char *dir, char *name, char *ext);
```

```
int _wfnsplit(const wchar_t *path, wchar_t *drive, wchar_t *dir, wchar_t *name, wchar_t *ext );
```

Description

Splits a full path name into its components.

fnsplit takes a file's full path name (path) as a string in the form X:\DIR\SUBDIR\NAME.EXT and splits path into its four components. It then stores those components in the strings pointed to by drive, dir, name, and ext. All five components must be passed but any of them can be a null which means the corresponding component will be parsed but not stored. If any path component is null, that component corresponds to a non-NULL, empty string.

The maximum sizes for these strings are given by the constants MAXDRIVE, MAXDIR, MAXPATH, MAXFILE, and MAXEXT (defined in dir.h) and each size includes space for the null-terminator.

fnsplit assumes that there is enough space to store each non-null component.

- When fnsplit splits path it treats the punctuation as follows:
- drive includes the colon (C:, A:, and so on)
- dir includes the leading and trailing backslashes (\BC\include\, \source\ ,and so on)
- name includes the file name
- ext includes the dot preceding the extension (.C, .EXE, and so on).

fnmerge and fnsplit are invertible; if you split a given path with fnsplit then merge the resultant components with fnmerge you end up with path.

Return Value

fnsplit returns an integer (composed of five flags defined in dir.h) indicating which of the full path name components were present in path. These flags and the components they represent are

EXTENSION	An extension
FILENAME	A file name
DIRECTORY	A directory (and possibly subdirectories)
DRIVE	A drive specification (see dir.h)
WILDCARDS	Wildcards (* or ?)

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <dir.h>
int main(void)
{
    char *s;
    char drive[MAXDRIVE];
```

```
    char dir[MAXDIR];
    char file[MAXFILE];
    char ext[MAXEXT];
    int flags;
    s=getenv("COMSPEC"); /* get the comspec environment parameter */
    flags=fnsplit(s,drive,dir,file,ext);
    printf("Command processor info:\n");
    if(flags & DRIVE)
       printf("\tdrive: %s\n",drive);
    if(flags & DIRECTORY)
       printf("\tdirectory: %s\n",dir);
    if(flags & FILENAME)
       printf("\tfile: %s\n",file);
    if(flags & EXTENSION)
       printf("\textension: %s\n",ext);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fnsplit		+		
_wfnsplit		NT only		

3.1.4.8.8 Bit Definitions for fnsplit

Header File

dir.h

Description

Bit definitions returned from fnsplit to identify which pieces of a file name were found during the split.

Flag	Component
DIRECTORY	Path includes a directory (and possibly subdirectories)
DRIVE	Path includes a drive specification (see DIR.H)
EXTENSION	Path includes an extension
FILENAME	Path includes a file name
WILDCARDS	Path contains wildcards (* or ?)

3.1.4.8.9 getcurdir, _wgetcurdir

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int getcurdir(int drive, char *directory);
```

```
int _wgetcurdir(int drive, wchar_t *directory );
```

Description

Gets current directory for specified drive.

getcurdir gets the name of the current working directory for the drive indicated by drive. drive specifies a drive number (0 for default, 1 for A, and so on). directory points to an area of memory of length MAXDIR where the null-terminated directory name will be placed. The name does not contain the drive specification and does not begin with a backslash.

Return Value

getcurdir returns 0 on success or -1 in the event of error.

Example

```
#include <dir.h>
#include <stdio.h>
#include <string.h>
char *current_directory(char *path)
{
  strcpy(path, "X:\\");      /* fill string with form of response: X:\ */
  path[0] = 'A' + getdisk();    /* replace X with current drive letter */
  getcurdir(0, path+3);  /* fill rest of string with current directory */
  return(path);
}
int main(void)
{
  char curdir[MAXPATH];
  current_directory(curdir);
  printf("The current directory is %s\n", curdir);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getcurdir		+		
_wgetcurdir		NT only		

3.1.4.8.10 getcwd, _wgetcwd

Header File

dir.h

Category

Directory Control Routines

Prototype

```
char *getcwd(char *buf, int buflen);
```

```
wchar_t *_wgetcwd(wchar_t *buf, int buflen);
```

Description

Gets current working directory.

getcwd gets the full path name (including the drive) of the current working directory, up to buflen bytes long and stores it in buf. If the full path name length (including the null terminator) is longer than buflen bytes, an error occurs.

If buf is NULL, a buffer buflen bytes long is allocated for you with malloc. You can later free the allocated buffer by passing the return value of getcwd to the function free.

Return Value

- getcwd returns the following values:
- If buf is not NULL on input, getcwd returns buf on success, NULL on error.
- If buf is NULL on input, getcwd returns a pointer to the allocated buffer.

In the event of an error return, the global variable errno is set to one of the following values:

ENODEV	No such device
ENOMEM	Not enough memory to allocate a buffer (buf is NULL)
ERANGE	Directory name longer than buflen (buf is not NULL)

Example

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
   char buffer[MAXPATH];
   getcwd(buffer, MAXPATH);
   printf("The current directory is: %s\n", buffer);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getcwd	+	+		
_wgetcwd		NT only		

3.1.4.8.11 getdisk, setdisk

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int getdisk(void);
```

```
int setdisk(int drive);
```

Description

Gets or sets the current drive number.

getdisk gets the current drive number. It returns an integer: 0 for A, 1 for B, 2 for C, and so on.

setdisk sets the current drive to the one associated with drive: 0 for A, 1 for B, 2 for C, and so on.

The setdisk function changes the current drive of the parent process.

Return Value

getdisk returns the current drive number. setdisk returns the total number of drives available.

Example

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
  int disk, maxdrives = setdisk(2);
  disk = getdisk() + 'A';
  printf("\nThe number of logical drives is:%d\n", maxdrives);
  printf("The current drive is: %c\n", disk);
  return 0;
 }
```

3.1.4.8.12 mkdir, _wmkdir

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int mkdir(const char *path);
```

```
int _wmkdir(const wchar_t *path);
```

Description

Creates a directory.

mkdir is available on UNIX, though it then takes an additional parameter.

mkdir creates a new directory from the given path name path.

Return Value

mkdir returns the value 0 if the new directory was created.

A return value of -1 indicates an error, and the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	No such file or directory

Example

```
#include <stdio.h>
#include <process.h>
#include <dir.h>
#define DIRNAME "testdir.$$$"
int main(void)
{
    int stat;
    stat = mkdir(DIRNAME);
    if (!stat)
           printf("Directory created\n");
    else
    {
       printf("Unable to create directory\n");
       exit(1);
    }
    getchar();
    system("dir/p");
    getchar();
    stat = rmdir(DIRNAME);
```

```
    if (!stat)
          printf("\nDirectory deleted\n");
    else
    {
          perror("\nUnable to delete directory\n");
       exit(1);
    }
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
mkdir	+	+		
_wmkdir		NT only		

3.1.4.8.13 _mktemp, _wmktemp

Header File

dir.h

Category

Directory Control Routines

Prototype

```
char *_mktemp(char *template);
```

```
wchar_t *_wmktemp(wchar_t *template);
```

Description

Makes a unique file name.

_mktemp replaces the string pointed to by template with a unique file name and returns template.

template should be a null-terminated string with six trailing Xs. These Xs are replaced with a unique collection of letters plus a period, so that there are two letters, a period, and three suffix letters in the new file name.

Starting with AA.AAA, the new file name is assigned by looking up the name on the disk and avoiding pre-existing names of the same format.

Return Value

If a unique name can be created and template is well formed, _mktemp returns the address of the template string. Otherwise, it returns null.

Example

```
#include <dir.h>
#include <stdio.h>
int main(void)
{
    /* fname defines the template for the
      temporary file.  */
    char *fname = "TXXXXXX", *ptr;
    ptr = mktemp(fname);
    printf("%s\n",ptr);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
mktemp	+	+		
_wmktemp		+		

3.1.4.8.14 _rmdir, _wrmdir

Header File

dir.h

Category

Directory Control Routines

Prototype

```
int _rmdir(const char *path);
```

```
int _wrmdir(const wchar_t *path);
```

Description

Removes a directory.

_rmdir deletes the directory whose path is given by path. The directory named by path

- must be empty
- must not be the current working directory
- must not be the root directory

Return Value

_rmdir returns 0 if the directory is successfully deleted. A return value of -1 indicates an error, and the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	Path or file function not found

Example

```
#include <stdio.h>
#include <process.h>
#include <dir.h>
#define DIRNAME "testdir.$$$"
int main(void)
{
    int stat;
    stat = mkdir(DIRNAME);
    if (!stat)
            printf("Directory created\n");
    else
    {
       printf("Unable to create directory\n");
       exit(1);
    }
    getchar();
    system("dir/p");
    getchar();
    stat = rmdir(DIRNAME);
```

```
    if (!stat)
          printf("\nDirectory deleted\n");
    else
    {
          perror("\nUnable to delete directory\n");
       exit(1);
    }
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rmdir	+	+		
_wrmdir		NT only		

3.1.4.8.15 searchpath, wsearchpath

Header File

dir.h

Category

Miscellaneous Routines

Prototype

```
char *searchpath(const char *file);
```

```
wchar_t *wsearchpath( const wchar_t *file );
```

Description

Searches the operating system path for a file.

searchpath attempts to locate file, searching along the operating system path, which is the PATH=... string in the environment. A pointer to the complete path-name string is returned as the function value.

searchpath searches for the file in the current directory of the current drive first. If the file is not found there, the PATH environment variable is fetched, and each directory in the path is searched in turn until the file is found, or the path is exhausted.

When the file is located, a string is returned containing the full path name. This string can be used in a call to access the file (for example, with fopen or exec...).

The string returned is located in a static buffer and is overwritten on each subsequent call to searchpath.

Return Value

searchpath returns a pointer to a file name string if the file is successfully located; otherwise, searchpath returns null.

Example

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
    char *p;
    /* Looks for ILINK32 and returns a pointer
       to the path  */
    p = searchpath("ILINK32.EXE");
    printf("Search for ILINK32.EXE : %s\n", p);
    /* Looks for nonexistent file  */
```

```
    p = searchpath("NOTEXIST.FIL");
    printf("Search for NOTEXIST.FIL : %s\n", p);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
searchpath		+		
wsearchpath		NT only		

3.1.4.9 dos.h

The following functions, macros, and classes are provided in `dos.h`:

Topics

Name	Description
FA_xxxx #defines (see page 822)	**Header File** dos.h **Description** File attributes
NFDS #define (see page 822)	**Header File** dos.h **Description** Maximum number of file descriptors.
_getdrive (see page 822)	**Header File** dos.h **Category** Directory Control Routines **Prototype** `int _getdrive(void);` **Description** Gets the current drive. _getdrive gets the the current drive number. It returns an integer: 0 for A, 1 for B, 2 for C, and so on. **Return Value** _getdrive returns the current drive number on success or -1 in the event of error. **Example**
_osmajor (see page 823)	**Header File** dos.h **Syntax** `extern unsigned char _osmajor;` **Description** The major version number of the operating system is available individually through _osmajor. For example, if you are running DOS version 3.2, _osmajor will be 3. This variable can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines behave differently depending on the DOS version number, while others only work under DOS 3.x and higher. For example, refer to creatnew and _rtl_open.

_osminor (see page 823)	**Header File** dos.h **Syntax** `extern unsigned char _osminor;` **Description** The minor version number of the operating system is available individually through _osminor. For example, if you are running DOS version 3.2, _osminor will be 20. This variables can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines behave differently depending on the DOS version number, while others only work under DOS 3.x and higher. For example, refer to creatnew and _rtl_open.
_osversion (see page 823)	**Header File** dos.h **Syntax** `extern unsigned _osversion;` **Description** _osversion contains the operating system version number, with the major version number in the low byte and the minor version number in the high byte. (For DOS version x.y, the x is the major version number, and y is the minor version number.) _osversion is functionally identical to _version.
_sleep (see page 824)	**Header File** dos.h **Category** Process Control Routines **Prototype** `void _sleep(unsigned seconds);` **Description** Suspends execution for an interval (seconds). With a call to _sleep, the current program is suspended from execution for the number of seconds specified by the argument seconds. The interval is accurate only to the nearest hundredth of a second or to the accuracy of the operating system clock, whichever is less accurate. **Return Value** None. **Example**
_version (see page 824)	**Header File** dos.h **Syntax** `extern unsigned _version;` **Description** _version contains the operating system version number, with the major version number in the low byte and the minor version number in the high byte. (For DOS version x.y, the x is the major version number, and y is the minor version number.)

disable, _disable, enable, _enable (see page 825)	**Header File**
	dos.h
	Category
	Miscellaneous Routines
	Prototype
	`void disable(void);`
	`void _disable(void);`
	`void enable(void);`
	`void _enable(void);`
	Description
	Disables and enables interrupts.
	These macros are designed to provide a programmer with flexible hardware interrupt control.
	disable and _disable macros disable interrupts. Only the NMI (non-maskable interrupt) is allowed from any external device.
	enable and _enable macros enable interrupts, allowing any device interrupts to occur.
	Return Value
	None.
	Portability

dostounix (see page 825)	**Header File** dos.h **Category** Time and Date Routines **Prototype** `long dostounix(struct date *d, struct time *t);` **Description** **Converts date and time to UNIX time format.** **dostounix converts a date and time as returned from getdate and gettime into UNIX time format. d points to a date structure, and t points to a time structure containing valid date and time information.** **The date and time must not be earlier than or equal to Jan 1 1980 00:00:00.** **Return Value** Returns UNIX version of current date and time parameters: number of seconds since 00:00:00 on January 1, 1970 (GMT). **Example**
geninterrupt (see page 826)	**Header File** **Category** **Prototype** `void geninterrupt(int intr_num);` **Description** **Return Value** **Portability**

getdate, setdate (see page 826)	**Header File**
	dos.h
	Category
	Time and Date Routines
	Prototype
	`void getdate(struct date *datep);`
	`void setdate(struct date *datep);`
	Description
	Gets and sets system date.
	getdate fills in the date structure (pointed to by datep) with the system's current date.
	setdate sets the system date (month, day, and year) to that in the date structure pointed to by datep. Note that a request to set a date might fail if you do not have the privileges required by the operating system.
	The date structure is defined as follows:
	`struct date{`
	`int da_year; /* current year */`
	`char da_day; /* day of the...  more (see page 826)`

getdfree (see page 827)	**Header File**
	dos.h
	Category
	Directory Control Routines, Miscellaneous Routines
	Prototype
	void getdfree(unsigned char drive, struct dfree *dtable);
	Description
	Gets disk free space.
	getdfree accepts a drive specifier in drive (0 for default, 1 for A, and so on) and fills the dfree structure pointed to by dtable with disk attributes.
	The dfree structure is defined as follows:
	`struct dfree {`
	`unsigned df_avail; /* available clusters */`
	`unsigned df_total; /* total clusters */`
	`unsigned df_bsec; /* bytes per sector */`
	`unsigned df_sclus; /* sectors per cluster */`
	`};`
	Return Value
	getdfree returns no value. In the event of an error, df_sclus... more (see page 827)

gettime, settime (see page 828)	**Header File**
	dos.h
	Category
	Time and Date Routines
	Prototype
	`void gettime(struct time *timep);`
	`void settime(struct time *timep);`
	Description
	Gets and sets the system time.
	gettime fills in the time structure pointed to by timep with the system's current time.
	• settime sets the system time to the values in the time structure pointed to by timep.
	The time structure is defined as follows:
	`struct time {`
	`unsigned char ti_min; /* minutes */`
	`unsigned char ti_hour; /* hours */`
	`unsigned char ti_hund; /* hundredths of seconds */`
	`unsigned char ti_sec; /* seconds */`
	`};`
	Return Value
	None.
	Example

unixtodos (see page 829)	**Header File** dos.h **Category** Time and Date Routines **Prototype** `void unixtodos(long time, struct date *d, struct time *t);` **Description** Converts date and time from UNIX to DOS format. unixtodos converts the UNIX-format time given in time to DOS format and fills in the date and time structures pointed to by d and t. time must not represent a calendar time earlier than Jan. 1, 1980 00:00:00. **Return Value** None. **Example**

_unlink, _wunlink (see page 830)	**Header File**
	dos.h
	Category
	Input/output Routines
	Prototype
	`int _unlink(const char *filename);`
	`int _wunlink(const wchar_t *filename);`
	Description
	Deletes a file.
	_unlink deletes a file specified by filename. Any drive, path, and file name can be used as a filename. Wildcards are not allowed.
	Read-only files cannot be deleted by this call. To remove read-only files, first use chmod or _rtl_chmod to change the read-only attribute.
	Note: If the file is open, it must be closed before unlinking it.
	_wunlink is the Unicode version of _wunlink. The Unicode version accepts a filename that is a wchar_t character string. Otherwise, the functions perform... more (see page 830)

3.1.4.9.1 FA_xxxx #defines

Header File

dos.h

Description

File attributes

Constant	Description
FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory
FA_ARCH	Archive

3.1.4.9.2 NFDS #define

Header File

dos.h

Description

Maximum number of file descriptors.

3.1.4.9.3 _getdrive

Header File

dos.h

Category

Directory Control Routines

Prototype

```
int _getdrive(void);
```

Description

Gets the current drive.

_getdrive gets the the current drive number. It returns an integer: 0 for A, 1 for B, 2 for C, and so on.

Return Value

_getdrive returns the current drive number on success or -1 in the event of error.

Example

```
#include <stdio.h>
#include <direct.h>
int main(void)
```

```
{
  int disk;
  disk = _getdrive() + 'A' - 1;
  printf("The current drive is: %c\n", disk);
  return 0;
}
```

3.1.4.9.4 _osmajor

Header File

dos.h

Syntax

```
extern unsigned char _osmajor;
```

Description

The major version number of the operating system is available individually through _osmajor. For example, if you are running DOS version 3.2, _osmajor will be 3.

This variable can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines behave differently depending on the DOS version number, while others only work under DOS 3.x and higher. For example, refer to creatnew and _rtl_open.

3.1.4.9.5 _osminor

Header File

dos.h

Syntax

```
extern unsigned char _osminor;
```

Description

The minor version number of the operating system is available individually through _osminor. For example, if you are running DOS version 3.2, _osminor will be 20.

This variables can be useful when you want to write modules that will run on DOS versions 2.x and 3.x. Some library routines behave differently depending on the DOS version number, while others only work under DOS 3.x and higher. For example, refer to creatnew and _rtl_open.

3.1.4.9.6 _osversion

Header File

dos.h

Syntax

```
extern unsigned _osversion;
```

Description

_osversion contains the operating system version number, with the major version number in the low byte and the minor version number in the high byte. (For DOS version x.y, the x is the major version number, and y is the minor version number.)

_osversion is functionally identical to _version.

3.1.4.9.7 _sleep

Header File

dos.h

Category

Process Control Routines

Prototype

```
void _sleep(unsigned seconds);
```

Description

Suspends execution for an interval (seconds).

With a call to _sleep, the current program is suspended from execution for the number of seconds specified by the argument seconds. The interval is accurate only to the nearest hundredth of a second or to the accuracy of the operating system clock, whichever is less accurate.

Return Value

None.

Example

```
#include <dos.h>
#include <stdio.h>
int main(void)
{
   int i;
   for (i=1; i<5; i++)
   {
      printf("Sleeping for %d seconds\n", i);
      _sleep(i);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.9.8 _version

Header File

dos.h

Syntax

```
extern unsigned _version;
```

Description

_version contains the operating system version number, with the major version number in the low byte and the minor version number in the high byte. (For DOS version x.y, the x is the major version number, and y is the minor version number.)

3.1.4.9.9 disable, _disable, enable, _enable

Header File

dos.h

Category

Miscellaneous Routines

Prototype

```
void disable(void);
void _disable(void);
void enable(void);
void _enable(void);
```

Description

Disables and enables interrupts.

These macros are designed to provide a programmer with flexible hardware interrupt control.

disable and _disable macros disable interrupts. Only the NMI (non-maskable interrupt) is allowed from any external device.

enable and _enable macros enable interrupts, allowing any device interrupts to occur.

Return Value

None.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.9.10 dostounix

Header File

dos.h

Category

Time and Date Routines

Prototype

```
long dostounix(struct date *d, struct time *t);
```

Description

Converts date and time to UNIX time format.

dostounix converts a date and time as returned from getdate and gettime into UNIX time format. d points to a date structure, and t points to a time structure containing valid date and time information.

The date and time must not be earlier than or equal to Jan 1 1980 00:00:00.

Return Value

Returns UNIX version of current date and time parameters: number of seconds since 00:00:00 on January 1, 1970 (GMT).

Example

```
#include <time.h>
#include <stddef.h>
#include <dos.h>
#include <stdio.h>
int main(void)
{
  time_t t;
  struct time d_time;
  struct date d_date;
  struct tm *local;
  getdate(&d_date);
  gettime(&d_time);
  t = dostounix(&d_date, &d_time);
  local = localtime(&t);
  printf("Time and Date: %s\n", asctime(local));
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.9.11 geninterrupt

Header File

Category

Prototype

```
void geninterrupt(int intr_num);
```

Description

Return Value

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.9.12 getdate, setdate

Header File

dos.h

Category

Time and Date Routines

Prototype

```
void getdate(struct date *datep);
```

```
void setdate(struct date *datep);
```

Description

Gets and sets system date.

getdate fills in the date structure (pointed to by datep) with the system's current date.

setdate sets the system date (month, day, and year) to that in the date structure pointed to by datep. Note that a request to set a date might fail if you do not have the privileges required by the operating system.

The date structure is defined as follows:

```
struct date{
int da_year; /* current year */
char da_day; /* day of the month */
char da_mon; /* month (1 = Jan) */
};
```

Return Value

getdate and setdate do not return a value.

Example

```
#include <dos.h>
#include <stdio.h>
int main(void)
{
   struct date d;
   getdate(&d);
   printf("The current year is: %d\n", d.da_year);
   printf("The current day is: %d\n", d.da_day);
   printf("The current month is: %d\n", d.da_mon);
   return 0;
}
```

3.1.4.9.13 getdfree

Header File

dos.h

Category

Directory Control Routines, Miscellaneous Routines

Prototype

```
void getdfree(unsigned char drive, struct dfree *dtable);
```

Description

Gets disk free space.

getdfree accepts a drive specifier in drive (0 for default, 1 for A, and so on) and fills the dfree structure pointed to by dtable with disk attributes.

The dfree structure is defined as follows:

```
struct dfree {
unsigned df_avail; /* available clusters */
unsigned df_total; /* total clusters */
```

```
unsigned df_bsec; /* bytes per sector */
unsigned df_sclus; /* sectors per cluster */
};
```

Return Value

getdfree returns no value. In the event of an error, df_sclus in the dfree structure is set to (**unsigned**) -1.

Example

```
#include <stdio.h>
#include <dos.h>
#include <process.h>
int main(void)
{
  struct dfree free;
  long avail;
  getdfree(0, &free);
  if ( free.df_sclus == -1)
  {
    printf("Error in getdfree() call\n");
    exit(1);
  }
  avail = (long) free.df_avail
        * (long) free.df_bsec
        * (long) free.df_sclus;
  printf("The current drive has %ld bytes available\n", avail);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.9.14 gettime, settime

Header File

dos.h

Category

Time and Date Routines

Prototype

```
void gettime(struct time *timep);
void settime(struct time *timep);
```

Description

Gets and sets the system time.

gettime fills in the time structure pointed to by timep with the system's current time.

- settime sets the system time to the values in the time structure pointed to by timep.

The time structure is defined as follows:

```
struct time {
unsigned char ti_min; /* minutes */
```

```
unsigned char ti_hour; /* hours */
unsigned char ti_hund; /* hundredths of seconds */
unsigned char ti_sec; /* seconds */
};
```

Return Value

None.

Example

```
#include   <stdio.h>
#include   <dos.h>
int main(void)
{
   struct  time t;
   gettime(&t);
   printf("The current time is: %2d:%02d:%02d.%02d\n",
          t.ti_hour, t.ti_min, t.ti_sec, t.ti_hund);
   return 0;
}
```

3.1.4.9.15 unixtodos

Header File

dos.h

Category

Time and Date Routines

Prototype

```
void unixtodos(long time, struct date *d, struct time *t);
```

Description

Converts date and time from UNIX to DOS format.

unixtodos converts the UNIX-format time given in time to DOS format and fills in the date and time structures pointed to by d and t.

time must not represent a calendar time earlier than Jan. 1, 1980 00:00:00.

Return Value

None.

Example

```
#include <stdio.h>
#include <dos.h>
char *month[] = {"---", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
#define SECONDS_PER_DAY 86400L  /* the number of seconds in one day */
struct date dt;
struct time tm;
int main(void)
{
   unsigned long val;
/* get today's date and time */
   getdate(&dt);
   gettime(&tm);
   printf("today is %d %s %d\n", dt.da_day, month[dt.da_mon], dt.da_year);
/*convert date and time to unix format (num of seconds since Jan 1, 1970*/
```

```
   val = dostounix(&dt, &tm);
/* subtract 42 days worth of seconds */
   val -= (SECONDS_PER_DAY * 42);
/* convert back to dos time and date */
   unixtodos(val, &dt, &tm);
   printf("42 days ago it was %d %s %d\n",
          dt.da_day, month[dt.da_mon], dt.da_year);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.9.16 _unlink, _wunlink

Header File

dos.h

Category

Input/output Routines

Prototype

```
int _unlink(const char *filename);
```

```
int _wunlink(const wchar_t *filename);
```

Description

Deletes a file.

_unlink deletes a file specified by filename. Any drive, path, and file name can be used as a filename. Wildcards are not allowed.

Read-only files cannot be deleted by this call. To remove read-only files, first use chmod or _rtl_chmod to change the read-only attribute.

Note: If the file is open, it must be closed before unlinking it.

_wunlink is the Unicode version of _wunlink. The Unicode version accepts a filename that is a wchar_t character string. Otherwise, the functions perform identically.

Return Value

On success, _unlink returns 0.

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
ENOENT	Path or file name not found

Example

```
#include <stdio.h>
#include <io.h>
int main(void)
{
   FILE *fp = fopen("junk.jnk","w");
   int status;
   fprintf(fp,"junk");
```

```
    status = access("junk.jnk",0);
    if (status == 0)
       printf("File exists\n");
    else
       printf("File doesn't exist\n");
    fclose(fp);
    unlink("junk.jnk");
    status = access("junk.jnk",0);
    if (status == 0)
       printf("File exists\n");
    else
       printf("File doesn't exist\n");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_unlink	+	+		
_wunlink		NT only		

3.1.4.10 errno.h

The following functions, macros, and classes are provided in `errno.h`:

Topics

Name	Description
EDOM, ERANGE, #defines (see page 832)	**Header File** errno.h, math.h **Description** These are the mnemonics and meanings for the error numbers found in math.h and errno.
_doserrno (see page 833)	**Header File** errno.h **Syntax** `extern int _doserrno;` **Description** _doserrno is a variable that maps many operating system error codes to errno; however, perror does not use _doserrno directly. When an operating system call results in an error, _doserrno is set to the actual operating system error code. errno is a parallel error variable inherited from UNIX. The following list gives mnemonics for the actual DOS error codes to which _doserrno can be set. (This value of _doserrno may or may not be mapped (through errno) to an equivalent error message string in _sys_errlist.
errno (C Runtime Library Reference) (see page 833)	**Header File** errno.h **Syntax** `extern int errno;` **Description** errno is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks. When an error in a math or system call occurs, errno is set to indicate the type of error. Sometimes errno and _doserrno are equivalent. At other times, errno does not contain the actual operating system error code, which is contained in _doserrno. Still other errors might occur that set only errno, not _doserrno.

Error Numbers in errno (see page 834)	**Header File** errno.h **Description** These are the mnemonics and meanings for the error numbers found in errno. Each value listed can be used to index into the sys_errlist array for displaying messages. Also, perror will display messages.

perror,_wperror (see page 835)	**Header File**
	errno.h, stdio.h
	Category
	Diagnostic Routines, Input/output Routines
	Prototype
	`void perror(const char *s);`
	`void _wperror(const wchar_t *s);`
	Description
	Prints a system error message.
	perror prints to the stderr stream (normally the console) the system error message for the last library routine that set the global variable errno.
	It prints the argument s followed by a colon (:) and the message corresponding to the current value of the global variable errno and finally a new line. The convention is to pass the file name of the program as the argument string.
	The array of error message strings is accessed through... more (see page 835)

_sys_errlist (see page 837)	**Header File**
	errno.h
	Syntax
	`extern char * _sys_errlist[ ];`
	Description
	_sys_errlist is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.
	To provide more control over message formatting, the array of message strings is provided in _sys_errlist. You can use errno as an index into the array to find the string corresponding to the error number. The string does not include any newline character.
	Example
	`printf("%s\n", _sys_errlist[ENOPATH]);`
	This code statement that uses the Mnemonic ENOPATH will output the string "Path not found".
	The following table gives mnemonics and their meanings for the... more (see page 837)

_sys_nerr (see page 838)	**Header File** errno.h **Syntax** `extern int _sys_nerr;` **Description** _sys_nerr is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks. This variable is defined as the number of error message strings in _sys_errlist.

3.1.4.10.1 EDOM, ERANGE, #defines

Header File

errno.h, math.h

Description

These are the mnemonics and meanings for the error numbers found in math.h and errno.

Name	Meaning
EDOM	Error code for math domain error
ERANGE	Error code for result out of range

3.1.4.10.2 _doserrno

Header File

errno.h

Syntax

```
extern int _doserrno;
```

Description

_doserrno is a variable that maps many operating system error codes to errno; however, perror does not use _doserrno directly.

When an operating system call results in an error, _doserrno is set to the actual operating system error code. errno is a parallel error variable inherited from UNIX.

The following list gives mnemonics for the actual DOS error codes to which _doserrno can be set. (This value of _doserrno may or may not be mapped (through errno) to an equivalent error message string in _sys_errlist.

E2BIG	Bad environ
EACCES	Access denied
EACCES	Bad access
EACCES	Is current dir
EBADF	Bad handle
EFAULT	Reserved
EINVAL	Bad data
EINVAL	Bad function
EMFILE	Too many open
ENOENT	No such file or directory
ENOEXEC	Bad format
ENOMEM	Mcb destroyed
ENOMEM	Out of memory
ENOMEM	Bad block
EXDEV	Bad drive
EXDEV	Not same device

3.1.4.10.3 errno (C Runtime Library Reference)

Header File

errno.h

Syntax

```
extern int errno;
```

Description

errno is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.

When an error in a math or system call occurs, errno is set to indicate the type of error. Sometimes errno and _doserrno are

equivalent. At other times, errno does not contain the actual operating system error code, which is contained in _doserrno. Still other errors might occur that set only errno, not _doserrno.

3.1.4.10.4 Error Numbers in errno

Header File

errno.h

Description

These are the mnemonics and meanings for the error numbers found in errno.

Each value listed can be used to index into the sys_errlist array for displaying messages.

Also, perror will display messages.

Mnemonic	Meaning
EZERO	Error 0
EINVFNC	Invalid function number
ENOFILE	File not found
ENOPATH	Path not found
ECONTR	Memory blocks destroyed
EINVMEM	Invalid memory block address
EINVENV	Invalid environment
EINVFMT	Invalid format
EINVACC	Invalid access code
EINVDAT	Invalid data
EINVDRV	Invalid drive specified
ECURDIR	Attempt to remove CurDir
ENOTSAM	Not same device
ENMFILE	No more files
ENOENT	No such file or directory
EMFILE	Too many open files
EACCES	Permission denied
EBADF	Bad file number
ENOMEM	Not enough memory
EFAULT	Unknown error
ENODEV	No such device
EINVAL	Invalid argument
E2BIG	Arg list too long
ENOEXEC	Exec format error
EXDEV	Cross-device link
ENFILE	Too many open files
ECHILD	No child process

ENOTTY	Terminal control function attempted on a file that is not a terminal. (POSIX – Not used in Win32 applications.)
ETXTBSY	Not used in Win32 applications
EFBIG	An attempt was made to write to a file, beyond the maximum file size. (POSIX – Not used in Win32 applications.)
ESOSPC	No space left on device
ESPIPE	Illegal seek
EROFS	Read-only file system
EMLINK	The number of links exceeds LINK_MAX. (POSIX – Not used in Win32 applications.)
EPIPE	Broken pipe
EDOM	Math argument
ERANGE	Result too large
EEXIST	File already exists
EDEADLOCK	Locking violation
EPERM	Operation not permitted
ESRCH	No such process id. (POSIX – Not used in Win32 applications.)
EINTR	Interrupted function call
EIO	Input/output error
ENXIO	No such device or address
EAGAIN	Resource temporarily unavailable
ENOTBLK	Not used in Win32 applications
EBUSY	Resource busy
ENOTDIR	A pathname component is not a directory. (POSIX – Not used in Win32 applications.)
EISDIR	An attempt was made to open a directory for writing, or to rename a file with the same name as an existing directory. (POSIX – Not used in Win32 applications.)
EUCLEAN	Not used in Win32 console applications

3.1.4.10.5 perror,_wperror

Header File

errno.h, stdio.h

Category

Diagnostic Routines, Input/output Routines

Prototype

```
void perror(const char *s);
void _wperror(const wchar_t *s);
```

Description

Prints a system error message.

perror prints to the stderr stream (normally the console) the system error message for the last library routine that set the global

variable errno.

It prints the argument s followed by a colon (:) and the message corresponding to the current value of the global variable errno and finally a new line. The convention is to pass the file name of the program as the argument string.

The array of error message strings is accessed through the global variable _sys_errlist. The global variable errno can be used as an index into the array to find the string corresponding to the error number. None of the strings include a newline character.

The global variable _sys_nerr contains the number of entries in the array.

The following messages are generated by perror:

Note: For Win32 GUI applications, stderr must be redirected.

Arg list too big

Attempted to remove current directory

Bad address

Bad file number

Block device required

Broken pipe

Cross-device link

Error 0

Exec format error

Executable file in use

File already exists

File too large

Illegal seek

Inappropriate I/O control operation

Input/output error

Interrupted function call

Invalid access code

Invalid argument Resource busy

Invalid dataResource temporarily unavailable

Invalid environment

Invalid format

Invalid function number

Invalid memory block address

Is a directory

Math argument

Memory arena trashed

Name too long

No child processes

No more files

No space left on device

No such device

No such device or address

No such file or directory

No such process

Not a directory

Not enough memory

Not same device

Operation not permitted

Path not found

Permission denied

Possible deadlock

Read-only file system

Resource busy

Resource temporarily unavailable

Result too large

Too many links

Too many open files

Example

```
#include <stdio.h>
int main(void)
{
    FILE *fp;
    fp = fopen("perror.dat", "r");
    if (!fp)
       perror("Unable to open file for reading");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
perror	+	+	+	+
_wperror		+		

3.1.4.10.6 _sys_errlist

Header File

errno.h

Syntax

```
extern char * _sys_errlist[ ];
```

Description

_sys_errlist is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.

To provide more control over message formatting, the array of message strings is provided in _sys_errlist. You can use errno as an index into the array to find the string corresponding to the error number. The string does not include any newline character.

Example

```
printf("%s\n", _sys_errlist[ENOPATH]);
```

This code statement that uses the Mnemonic ENOPATH will output the string "Path not found".

The following table gives mnemonics and their meanings for the values stored in _sys_errlist. The list is alphabetically ordered for ease your reading convenience. For the numerical ordering, see the header file errno.h.

3.1.4.10.7 _sys_nerr

Header File

errno.h

Syntax

```
extern int _sys_nerr;
```

Description

_sys_nerr is used by perror to print error messages when certain library routines fail to accomplish their appointed tasks.

This variable is defined as the number of error message strings in _sys_errlist.

3.1.4.11 except.h

The following functions, macros, and classes are provided in `except.h`:

Topics

Name	Description

__throwExceptionName (see page 840)	**Header File**
	except.h
	Syntax
	`extern char * _RTLENTRY __ThrowExceptionName();`
	`#define __throwExceptionName __ThrowExceptionName()`
	Description
	Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.

__throwFileName (see page 840)	**Header File**
	except.h
	Syntax
	`extern char * _RTLENTRY __ThrowFileName();`
	`#define __throwFileName __ThrowFileName()`
	Description
	Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.
	To get the file name for a thrown exception with __throwFileName, you must compile the module with the -xp compiler option.

__throwLineNumber (see page 840)	**Header File**
	except.h
	Syntax
	`extern unsigned _RTLENTRY __ThrowLineNumber();`
	`#define __throwLineNumber __ThrowLineNumber()`
	Description
	Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.
	To get the line number for a thrown exception with __throwLineNumber, you must compile the module with the -xp compiler option.

set_terminate (see page 840)	**Header File**
	except.h
	Syntax
	`typedef void (*terminate_handler)();`
	`terminate_handler set_terminate(terminate_handler t_func);`
	Description
	set_terminate lets you install a function that defines the program's termination behavior when a handler for the exception cannot be found. The actions are defined in t_func, which is declared to be a function of type terminate_handler. A terminate_handler type, defined in except.h, is a function that takes no arguments, and returns void.
	By default, an exception for which no handler can be found results in the program calling the terminate function. This will normally result in a call to abort. The program then ends with the message Abnormal... more (see page 840)

set_unexpected (see page 841)	**Header File**
	except.h
	Syntax
	`typedef void ( * unexpected_handler )();`
	`unexpected_handler set_unexpected(unexpected_handler unexpected_func);`
	Description
	set_unexpected lets you install a function that defines the program's behavior when a function throws an exception not listed in its exception specification. The actions are defined in unexpected_func, which is declared to be a function of type unexpected_handler. An unexpected_handler type, defined in except.h, is a function that takes no arguments, and returns void.
	By default, an unexpected exception causes unexpected to be called. If is defined, it is subsequently called by unexpected. Program control is then turned over to the user-defined unexpected_func. Otherwise, terminate... more (see page 841)

terminate (see page 841)	**Header File** except.h **Syntax** `void terminate();` **Description** The function terminate can be called by unexpected or by the program when a handler for an exception cannot be found. The default action by terminate is to call abort. Such a default action causes immediate program termination. You can modify the way that your program will terminate when an exception is generated that is not listed in the exception specification. If you do not want the program to terminate with a call to abort, you can instead define a function to be called. Such a function (called a terminate_handler) will be called... more (see page 841)
unexpected (see page 842)	**Header File** except.h **Syntax** `void unexpected();` **Description** The unexpected function is called when a function throws an exception not listed in its exception specification. The program calls unexpected, which by default calls any user-defined function registered by set_unexpected. If no function is registered with set_unexpected, the unexpected function then calls terminate. **Return Value** None, although unexpected may throw an exception.

3.1.4.11.1 __throwExceptionName

Header File

except.h

Syntax

```
extern char * _RTLENTRY __ThrowExceptionName();
#define __throwExceptionName __ThrowExceptionName()
```

Description

Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.

3.1.4.11.2 __throwFileName

Header File

except.h

Syntax

```
extern char * _RTLENTRY __ThrowFileName();
#define __throwFileName __ThrowFileName()
```

Description

Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.

To get the file name for a thrown exception with __throwFileName, you must compile the module with the -xp compiler option.

3.1.4.11.3 __throwLineNumber

Header File

except.h

Syntax

```
extern unsigned _RTLENTRY __ThrowLineNumber();
#define __throwLineNumber __ThrowLineNumber()
```

Description

Use this global variable to get the name of a thrown exception. The output for this variable is a printable character string.

To get the line number for a thrown exception with __throwLineNumber, you must compile the module with the -xp compiler option.

3.1.4.11.4 set_terminate

Header File

except.h

Syntax

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler t_func);
```

Description

set_terminate lets you install a function that defines the program's termination behavior when a handler for the exception cannot be found. The actions are defined in t_func, which is declared to be a function of type terminate_handler. A terminate_handler type, defined in except.h, is a function that takes no arguments, and returns void.

By default, an exception for which no handler can be found results in the program calling the terminate function. This will normally result in a call to abort. The program then ends with the message Abnormal program termination. If you want some function other than abort to be called by the terminate function, you should define your own t_func function. Your t_func function is installed by set_terminate as the termination function. The installation of t_func lets you implement any actions that are not taken by abort.

Return Value

The previous function given to set_terminate will be the return value.

The definition of t_func must terminate the program. Such a user-defined function must not return to its caller, the terminate function. An attempt to return to the caller results in undefined program behavior. It is also an error for t_func to throw an exception.

3.1.4.11.5 set_unexpected

Header File

except.h

Syntax

```
typedef void ( * unexpected_handler )();
unexpected_handler set_unexpected(unexpected_handler unexpected_func);
```

Description

set_unexpected lets you install a function that defines the program's behavior when a function throws an exception not listed in its exception specification. The actions are defined in unexpected_func, which is declared to be a function of type unexpected_handler. An unexpected_handler type, defined in except.h, is a function that takes no arguments, and returns void.

By default, an unexpected exception causes unexpected to be called. If is defined, it is subsequently called by unexpected. Program control is then turned over to the user-defined unexpected_func. Otherwise, terminate is called.

Return Value

The previous function given to set_unexpected will be the return value.

The definition of unexpected_func must not return to its caller, the unexpected function. An attempt to return to the caller results in undefined program behavior.

unexpected_func can also call abort, exit, or terminate.

3.1.4.11.6 terminate

Header File

except.h

Syntax

```
void terminate();
```

Description

The function terminate can be called by unexpected or by the program when a handler for an exception cannot be found. The default action by terminate is to call abort. Such a default action causes immediate program termination.

You can modify the way that your program will terminate when an exception is generated that is not listed in the exception specification. If you do not want the program to terminate with a call to abort, you can instead define a function to be called. Such a function (called a terminate_handler) will be called by terminate if it is registered with set_terminate.

Return Value

None.

3.1.4.11.7 unexpected

Header File

except.h

Syntax

```
void unexpected();
```

Description

The unexpected function is called when a function throws an exception not listed in its exception specification. The program calls unexpected, which by default calls any user-defined function registered by set_unexpected. If no function is registered with set_unexpected, the unexpected function then calls terminate.

Return Value

None, although unexpected may throw an exception.

3.1.4.12 fastmath.h

The following functions, macros, and classes are provided in `fastmath.h`:

Topics

Name	Description
Using fastmath math routines (see page 842)	**Header File** fastmath.h **Category** Math Routines **Description** The FastMath routines are high performance math routines that don't check for most error conditions and never call matherr. They are coded for maximum speed. These functions are never exported from the RTLDLL, which means that they always get linked directly into the PE file that is being created. When you include fastmath.h, the following math functions are remapped to these fastmath functions.

3.1.4.12.1 Using fastmath math routines

Header File

fastmath.h

Category

Math Routines

Description

The FastMath routines are high performance math routines that don't check for most error conditions and never call matherr. They are coded for maximum speed. These functions are never exported from the RTLDLL, which means that they always get linked directly into the PE file that is being created.

When you include fastmath.h, the following math functions are remapped to these fastmath functions.

Math Routine	Fastmath Routine	Math Routine	Fastmath Routine
acos	_fm_acos	asin	_fm_asin
atan	_fm_atan	atan2	_fm_atan2
cos	_fm_cos	cosh	_fm_cosh
exp	_fm_exp	fabs	_fm_fabs
asin	_fm_asin	atan	_fm_atan
atan2	_fm_atan2	cos	_fm_cos
cosh	_fm_cosh	exp	_fm_exp
fabs	_fm_fabs	frexp	_fm_frexp
hypot	_fm_hypot	ldexp	_fm_ldexp
log	_fm_log	log10	_fm_log10
sin	_fm_sin	sinh	_fm_sinh
sqrt	_fm_sqr	tan	_fm_tan
tanh	_fm_tanh	sincos	_fm_sincos
acosl	_fm_acosl	asinl	_fm_asinl
atan2l	_fm_atan2l	atanl	_fm_atanl
coshl	_fm_coshl	cosl	_fm_cosl
expl	_fm_expl	fabsl	_fm_fabsl
frexpl	_fm_frexpl	hypotl	_fm_hypotl
ldexpl	_fm_ldexpl	log10l	_fm_log10l
logl	_fm_logl	sinhl	_fm_sinhl
sinl	_fm_sinl	sqrtl	_fm_sqrtl
tanhl	_fm_tanhl	tanl	_fm_tanl
sincosl	_fm_sincosl	atanhl	_fm_atanhl
acoshl	_fm_acoshl	asinhl	_fm_asinhl

If you don't want the standard C function names remapped to the FastMath versions, then define _FM_NO_REMAP. The FastMath routines can still be called with their _fm_xxx names.

The following additional functions are available in FastMath; they are not directly supported in the regular RTL:

```
void _FMAPI _fm_sincos(double __a, double *__x, double *__y);
void _FMAPI _fm_sincosl(long double __a, long double *__x, long double *__y);
long double _FMAPI _fm_atanhl (long double __x);
```

```
long double _FMAPI _fm_acoshl (long double __x);
long double _FMAPI _fm_asinhl (long double __x);
__inline void _fm_fwait(void)
unsigned int _FMAPI _fm_init(void);
```

_fm_fwait is a special inline function that performs an intrinsic FWAIT instruction.

_Fm_init is a function that can be called to mask all fpu exceptions prior to using the FastMath routines.

3.1.4.13 fcntl.h

The following functions, macros, and classes are provided in `fcntl.h`:

Topics

Name	Description
O_xxxx #defines (see page 845)	**Header File** fcntl.h **Description** These #defines are bit definitions for a file-access argument. These RTL file-open functions use some (not all) of these definitions: • fdopen • fopen • freopen • _fsopen • open • _rtl_open • sopen sopen also uses file-sharing symbolic constants in the file-access argument.
_fmode (see page 847)	**Header File** fcntl.h **Syntax** `extern int _fmode;` **Description** _fmode determines in which mode (text or binary) files will be opened and translated. The value of _fmode is O_TEXT by default, which specifies that files will be read in text mode. If _fmode is set to O_BINARY, the files are opened and read in binary mode. (O_TEXT and O_BINARY are defined in fcntl.h.) In text mode, carriage-return/linefeed (CR/LF) combinations are translated to a single linefeed character (LF) on input. On output, the reverse is true: LF characters are translated to CR/LF combinations. In binary mode, no such translation occurs. You can... more (see page 847)

_pipe (see page 847)	**Header File** io.h, fcntl.h **Category** Input/output Routines **Syntax** `int _pipe(int *handles, unsigned int size, int mode);` **Description** Creates a read/write pipe. The _pipe function creates an anonymous pipe that can be used to pass information between processes. The pipe is opened for both reading and writing. Like a disk file, a pipe can be read from and written to, but it does not have a name or permanent storage associated with it; data written to and from the pipe exist only in a memory buffer managed by the operating system. The read handle is returned to handles[0], and the write... more (see page 847)

open, _wopen (see page 849)	**Header File**
	io.h, fcntl.h
	Category
	Input/output Routines
	Prototype
	`int open(const char *path, int access [, unsigned mode]);`
	`int _wopen(const wchar_t *path, int access [, unsigned mode]);`
	Description
	Opens a file for reading or writing.
	open opens the file specified by path, then prepares it for reading and/or writing as determined by the value of access.
	To create a file in a particular mode, you can either assign to the global variable _fmode or call open with the O_CREAT and O_TRUNC options ORed with the translation mode desired.
	`open("XMP",O_CREAT|O_TRUNC|O_BINARY,S_IREAD)`
	creates a binary-mode, read-only file named XMP, truncating its length to 0... more (see page 849)

_sopen, _wsopen (see page 850)	**Header File**
	fcntl.h, sys\stat.h, share.h, io.h, stdio.h
	Category
	Input/output Routines
	Prototype
	`int _sopen(char *path, int access, int shflag[, int mode]);`
	`int _wsopen(wchar_t *path, int access, int shflag[, int mode]);`
	Description
	Opens a shared file.
	_sopen opens the file given by path and prepares it for shared reading or writing, as determined by access, shflag, and mode.
	_wsopen is the Unicode version of _sopen. The Unicode version accepts a filename that is a wchar_t character string. Otherwise, the functions perform identically.
	For _sopen, access is constructed by ORing flags bitwise from the following lists:
	Read/write flags
	You can use only one... more (see page 850)

3.1.4.13.1 O_xxxx #defines

Header File

fcntl.h

Description

These #defines are bit definitions for a file-access argument.

These RTL file-open functions use some (not all) of these definitions:

- fdopen
- fopen

- freopen
- _fsopen
- open
- _rtl_open
- sopen

sopen also uses file-sharing symbolic constants in the file-access argument.

	Constant	Description
Read/Write flag		
	O_RDONLY	Open for reading only
	O_WRONLY	Open for writing only
	O_RDWR	Open for reading and writing
Other access flags		
	O_NDELAY	Not used; for UNIX compatibility.
	O_APPEND	Append to end of file
		If set, the file pointer is set to the end of the file prior to each write.
	O_CREAT	Create and open file
		If the file already exists, has no effect.
		If the file does not exist, the file is created.
	O_EXCL	Exclusive open: Used only with O_CREAT.
		If the file already exists, an error is returned.
	O_TRUNC	Open with truncation
		If the file already exists, its length is truncated to 0. The file attributes remain unchanged.
Binary-mode/Text-mode flags		
	O_BINARY	No translation: Explicitly opens the file in binary mode
	O_TEXT	CR-LF translation: Explicitly opens the file in text mode
Additional values available using _rtl_open		
	O_NOINHERIT	Child processes inherit file
	O_DENYALL	Error if opened for read/write
	O_DENYWRITE	Error if opened for write
	O_DENYREAD	Error if opened for read
	O_DENYNONE	Allow concurrent access

Note: Only one of the O_DENYxxx options can be included in a single open. These file-sharing attributes are in addition to any locking performed on the files.

Do not modify		
	O_CHANGED	Special DOS read-only bit
	O_DEVICE	Special DOS read-only bit

3.1.4.13.2 _fmode

Header File

fcntl.h

Syntax

```
extern int _fmode;
```

Description

_fmode determines in which mode (text or binary) files will be opened and translated. The value of _fmode is O_TEXT by default, which specifies that files will be read in text mode. If _fmode is set to O_BINARY, the files are opened and read in binary mode. (O_TEXT and O_BINARY are defined in fcntl.h.)

In text mode, carriage-return/linefeed (CR/LF) combinations are translated to a single linefeed character (LF) on input. On output, the reverse is true: LF characters are translated to CR/LF combinations.

In binary mode, no such translation occurs.

You can override the default mode as set by _fmode by specifying a t (for text mode) or b (for binary mode) in the argument type in the library functions fopen, fdopen, and freopen. Also, in the function open, the argument access can include either O_BINARY or O_TEXT, which will explicitly define the file being opened (given by the path argument to the open function) to be in either binary or text mode.

3.1.4.13.3 _pipe

Header File

io.h, fcntl.h

Category

Input/output Routines

Syntax

```
int _pipe(int *handles, unsigned int size, int mode);
```

Description

Creates a read/write pipe.

The _pipe function creates an anonymous pipe that can be used to pass information between processes. The pipe is opened for both reading and writing. Like a disk file, a pipe can be read from and written to, but it does not have a name or permanent storage associated with it; data written to and from the pipe exist only in a memory buffer managed by the operating system.

The read handle is returned to handles[0], and the write handle is returned to handles[1]. The program can use these handles in subsequent calls to read, write, dup, dup2, or close. When all pipe handles are closed, the pipe is destroyed.

The size of the internal pipe buffer is size. A recommended minimum value is 512 bytes.

The translation mode is specified by mode, as follows:

O_BINARY	The pipe is opened in binary mode
O_TEXT	The pipe is opened in text mode

If mode is zero, the translation mode is determined by the external variable _fmode.

Return Value

On success, _pipe returns 0 and returns the pipe handles to handles[0] and handles[1].

On error, it returns -1 and sets errno to one of the following values:

EMFILE	Too many open files
ENOMEM	Out of memory

Example

```
/*
 There are two short programs here.  SEND spawns a child
 process, RECEIVE.  Each process holds one end of a
 pipe.  The parent transmits its command-line argument
 to the child, which prints the string and exits.
 IMPORTANT:  The parent process must be linked with
 the \32bit\fileinfo.obj file.  The code in fileinfo
 enables a parent to share handles with a child.
 Without this extra information, the child cannot use
 the handle it receives.
*/
/* SEND */
#include <fcntl.h>            // _pipe()
#include <io.h>             // write()
#include <process.h>          // spawnl() cwait()
#include <stdio.h>           // puts() perror()
#include <stdlib.h>           // itoa()
#include <string.h>           // strlen()
#define DECIMAL_RADIX 10      // for atoi()
enum PIPE_HANDLES { IN, OUT };  // to index the array of handles
int main(int argc, char *argv[])
{
  int handles[2];           // in- and
//outbound pipe handles
  char handleStr[10];         // a handle
//stored as a string
  int pid;
  // system's ID for child process
  if (argc <= 1)
  {
    puts("No message to send.");
    return(1);
  }
  if (_pipe(handles, 256, O_TEXT) != 0)
  {
    perror("Cannot create the pipe");
    return(1);
  }
  // store handle as a string for passing on the command line
  itoa(handles[IN], handleStr, DECIMAL_RADIX);
  // create the child process, passing it the inbound pipe handle
     spawnl(P_NOWAIT, "receive.exe", "receive.exe", handleStr, NULL);
  // transmit the message
  write(handles[OUT], argv[1], strlen(argv[1])+1);
  // when done with the pipe, close both handles
  close(handles[IN]);
  close(handles[OUT]);
```

```
  // wait for the child to finish
  wait(NULL);
  return(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.13.4 open, _wopen

Header File

io.h, fcntl.h

Category

Input/output Routines

Prototype

```
int open(const char *path, int access [, unsigned mode]);
```

```
int _wopen(const wchar_t *path, int access [, unsigned mode]);
```

Description

Opens a file for reading or writing.

open opens the file specified by path, then prepares it for reading and/or writing as determined by the value of access.

To create a file in a particular mode, you can either assign to the global variable _fmode or call open with the O_CREAT and O_TRUNC options ORed with the translation mode desired.

```
open("XMP",O_CREAT|O_TRUNC|O_BINARY,S_IREAD)
```

creates a binary-mode, read-only file named XMP, truncating its length to 0 bytes if it already existed.

For open, access is constructed by bitwise ORing flags from the following lists. Only one flag from the first list can be used (and one must be used); the remaining flags can be used in any logical combination.

These symbolic constants are defined in fcntl.h.

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

O_NDELAY	Not used; for UNIX compatibility.
O_APPEND	If set, the file pointer will be set to the end of the file prior to each write.
O_CREAT	If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of mode are used to set the file attribute bits as in chmod.
O_TRUNC	If the file exists, its length is truncated to 0. The file attributes remain unchanged.

O_EXCL	Used only with O_CREAT. If the file already exists, an error is returned.
O_BINARY	Can be given to explicitly open the file in binary mode.
O_TEXT	Can be given to explicitly open the file in text mode.

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable _fmode.

If the O_CREAT flag is used in constructing access, you need to supply the mode argument to open from the following symbolic constants defined in sys\stat.h.

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD|S_IWRITE	Permission to read and write

Return Value

On success, open returns a nonnegative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of the file.

On error, open returns -1 and the global variable errno is set to one of the following values:

EACCES	Permission denied
EINVACC	Invalid access code
EMFILE	Too many open files
ENOENT	No such file or directory

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "Hello world";
   if ((handle = open("TEST.$$$", O_CREAT | O_TEXT)) == -1)
   {
      perror("Error:");
      return 1;
   }
   write(handle, msg, strlen(msg));
   close(handle);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
open	+	+		
_wopen		NT only		

3.1.4.13.5 _sopen, _wsopen

Header File

fcntl.h, sys\stat.h, share.h, io.h, stdio.h

Category

Input/output Routines

Prototype

```
int _sopen(char *path, int access, int shflag[, int mode]);
int _wsopen(wchar_t *path, int access, int shflag[, int mode]);
```

Description

Opens a shared file.

_sopen opens the file given by path and prepares it for shared reading or writing, as determined by access, shflag, and mode.

_wsopen is the Unicode version of _sopen. The Unicode version accepts a filename that is a wchar_t character string. Otherwise, the functions perform identically.

For _sopen, access is constructed by ORing flags bitwise from the following lists:

Read/write flags

You can use only one of the following flags:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

Other access flags

You can use any logical combination of the following flags:

O_NDELAY	Not used; for UNIX compatibility.
O_APPEND	If set, the file pointer is set to the end of the file prior to each write.
O_CREA	If the file exists, this flag has no effect. If the file does not exist, the file is created, and the bits of mode are used to set the file attribute bits as in chmod.
O_TRUNC	If the file exists, its length is truncated to 0. The file attributes remain unchanged.
O_EXCL	Used only with O_CREAT. If the file already exists, an error is returned.
O_BINARY	This flag can be given to explicitly open the file in binary mode.
O_TEXT	This flag can be given to explicitly open the file in text mode.
O_NOINHERIT	The file is not passed to child programs.

Note: These O_... symbolic constants are defined in fcntl.h.

If neither O_BINARY nor O_TEXT is given, the file is opened in the translation mode set by the global variable _fmode.

If the O_CREAT flag is used in constructing access, you need to supply the mode argument to _sopen from the following symbolic constants defined in sys\stat.h.

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD|S_IWRITE	Permission to read/write

shflag specifies the type of file-sharing allowed on the file path. Symbolic constants for shflag are defined in share.h.

SH_COMPAT	Sets compatibility mode.
SH_DENYRW	Denies read/write access
SH_DENYWR	Denies write access
SH_DENYRD	Denies read access
SH_DENYNONE	Permits read/write access
SH_DENYNO	Permits read/write access

Return Value

On success, _sopen returns a nonnegative integer (the file handle), and the file pointer (that marks the current position in the file) is set to the beginning of the file.

On error, it returns -1, and the global variable errno is set to

EACCES	Permission denied
EINVACC	Invalid access code
EMFILE	Too many open files
ENOENT	Path or file function not found

Example

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   int handle,
       handle1;
   handle = sopen("c:\\autoexec.bat", O_RDONLY, SH_DENYWR, S_IREAD);
   if      (handle == -1)
   {
     perror (sys_errlist[errno]);
     exit (1);
   }
   if (!handle)
   {
      printf("sopen failed\n");
      exit(1);
   }
   /*      Attempt sopen for write.
   */
   handle1 = sopen("c:\\autoexec.bat", O_RDONLY, SH_DENYWR, S_IREAD);
   if      (handle1 == -1)
   {
     perror (sys_errlist[errno]);
     exit (1);
   }
   if (!handle1)
   {
      printf("sopen failed\n");
      exit(1);
   }
   close (handle);
   close (handle1);
```

```
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_sopen		+		
_wsopen		+		

3.1.4.14 float.h

The following functions, macros, and classes are provided in `float.h`:

Topics

Name	Description
CW_DEFAULT #define (see page 856)	**Header File** float.h **Description** Default control word for 80x87 math coprocessor.

_chgsign, _chgsignl (see page 856)	**Header File**
	float.h
	Category
	Math Routintes
	Prototype
	`double _chgsign(double d);`
	`long double _chgsignl(long double ld);`
	Description
	Reverses the sign of a double-precision floating-point argument, d.
	_chgsignl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	Returns a value of the same magnitude and exponent as the argument, but with the opposite sign. There is no error return value.
	Portability

_clear87, _clearfp (see page 857)	**Header File**
	float.h
	Category
	Math Routines
	Prototype
	`unsigned int _clear87 (void);`
	`unsigned int _clearfp (void);`
	Description
	Clears the floating-point status word.
	_clear87 clears the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by the 80x87 exception handler.
	_clearfp is identical to _clear87 and is for Microsoft compatibility.
	Return Value
	The bits in the value returned indicate the floating-point status before it was cleared. For information on the status word, refer to the constants defined in float.h.
	Example

_control87, _controlfp (see page 858)	**Header File**
	float.h
	Category
	Math Routines
	Prototype
	`unsigned int _control87(unsigned int newcw, unsigned int mask);`
	`unsigned int _controlfp(unsigned int newcw, unsigned int mask);`
	Description
	Manipulates the floating-point control word.
	_control87 retrieves or changes the floating-point control word.
	The floating-point control word is an **unsigned int** that, bit by bit, specifies certain modes in the floating-point package; namely, the precision, infinity, and rounding modes. Changing these modes lets you mask or unmask floating-point exceptions.
	_control87 matches the bits in mask to the bits in newcw. If a mask bit equals 1, the corresponding bit in newcw contains the new value... more (see page 858)

_copysign, _copysignl (see page 859)	**Header File**
	float.h
	Category
	Math Routines
	Prototype
	`double _copysign(double da, double db);`
	`long double _copysignl(long double lda, long double ldb);`
	Description
	Returns the double-precision floating point argument da, with the same sign as the double-precision floating-point argument db.
	_copysignl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	Returns the first value with the same magnitude and exponent, but with the sign of the second value. There is no error value returned.
	Portability

_finite, _finitel (see page 859)	**Header File**
	float.h
	Category
	Math Routines
	Prototype
	`int _finite(double d);`
	`int _finitel(long double ld);`
	Description
	Determines whether a given double-precision floating point value d is finite.
	_finitel is the **long double** version; it takes a **long double** argument.
	Return Value
	Returns non-zero if the argument is finite, and 0 if it is not.
	Portability

_fpclass, _fpclassl (see page 860)	**Header File**
	float.h
	Category
	Math Routines
	Prototype
	`int _fpclass(double d);`
	`int _fpclassl(long double ld);`
	Description
	Returns an integer value representing the type (class) of an IEEE real for doubles. This value contains information on the floating-point class of the argument.
	_fpclassl is the **long double** version; it takes a **long double** argument and returns the type (class) of an IEEE real for long doubles.
	Return Value
	Returns an integer value that indicates the floating-point class of its argument. The possible values, which are listed in the table below, are defined in FLOAT.H.
	Portability

_fpreset (see page 860)	**Header File** float.h **Category** Math Routines **Prototype** `void _fpreset(void);` **Description** Reinitializes floating-point math package. _fpreset reinitializes the floating-point math package. This function is usually used in conjunction with system or the exec... or spawn... functions. It is also used to recover from floating-point errors before calling longjmp. **Note:** If an 80x87 coprocessor is used in a program a child process (executed by the system, or by an exec... or spawn... function) might alter the parent process' floating-point state. • If you use an 80x87 take the following precautions: • Do not call system or an exec... or spawn... function while a... more (see page 860)

_isnan, _isnanl (see page 861)	**Header File**
	float.h
	Category
	Classification Routines, Math Routines
	Prototype
	`int _isnan(double d);`
	`int _isnanl(long double ld);`
	Description
	Tests whether a given double-precision floating-point value d is a NaN.
	_isnanl is the **long double** version; it takes a **long double** argument.
	Return Value
	Returns a nonzero value (TRUE) if the value passed in is a NaN; otherwise it returns 0 (FALSE). The non-zero return value corresponds to either _FPCLASS_SNAN, if the NaN is of the signaling type, or _FPCLASS_QNAN, if the NaN is of the quiet type. The values for _FPCLASS_SNAN and _FPCLASS_QNAN are in float.h.
	Portability

_logb, _logbl (see page 862)	**Header File**
	float.h
	Category
	Math Routines
	Prototype
	`double _logb(double d);`
	`long double _logbl(long double ld);`
	Description
	Extracts the exponential value of a double-precision floating-point argument. If the argument is denormalized, it is treated as if it were normalized.
	_logbl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	Returns the unbiased exponent of the value passed in.
	Portability

_nextafter, _nextafterl (see page 862)	**Header File**
	float.h
	Category
	Math Routines
	Prototype
	`double _nextafter(double da, double db);`
	`long double _nextafterl(long double lda, long double ldb);`
	Description
	Takes two doubles (da and db) and returns the closest representable neighbor of da in the direction toward db.
	If da == db, _nextafter returns da, with no exception triggered. If either da or db is a quiet NaN, then the return value is one or the other of the input NaNs.
	_nextafterl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	Returns the closest representable neighbor of... more (see page 862)

_scalb, _scalbl (see page 863)	**Header File**
	float.h
	Category
	Math Routines
	Prototype
	`double _scalb(double d, long exp);`
	`long double _scalbl(long double ld, long exp);`
	Description
	Scales the argument d by a power of 2.
	_scalbl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	Returns an exponential value if successful. On overflow (depending on the sign of the argument), the function returns +/– HUGE_VAL; the ERRNO variable is set to ERANGE.
	Portability

_status87, _statusfp (see page 863)	**Header File**
	float.h
	Category
	Math Routines
	Prototype
	`unsigned int _status87(void);`
	`unsigned int _statusfp(void);`
	Description
	Gets floating-point status.
	_status87 gets the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by the 80x87 exception handler.
	_statusfp is identical to _status87 and is for Microsoft compatibility.
	Return Value
	The bits in the return value give the floating-point status. See float.h for a complete definition of the bits returned by _status87 and _status87.
	Portability

3.1.4.14.1 CW_DEFAULT #define

Header File

float.h

Description

Default control word for 80x87 math coprocessor.

3.1.4.14.2 _chgsign, _chgsignl

Header File

float.h

Category

Math Routintes

Prototype

```
double _chgsign(double d);
long double _chgsignl(long double ld);
```

Description

Reverses the sign of a double-precision floating-point argument, d.

_chgsignl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns a value of the same magnitude and exponent as the argument, but with the opposite sign. There is no error return value.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_chgsign		+		
_chgsignl		+		

3.1.4.14.3 _clear87, _clearfp

Header File

float.h

Category

Math Routines

Prototype

```
unsigned int _clear87 (void);
unsigned int _clearfp (void);
```

Description

Clears the floating-point status word.

_clear87 clears the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by the 80x87 exception handler.

_clearfp is identical to _clear87 and is for Microsoft compatibility.

Return Value

The bits in the value returned indicate the floating-point status before it was cleared. For information on the status word, refer to the constants defined in float.h.

Example

```
#include <stdio.h>
#include <float.h>
int main(void)
{
```

```
    float x;
    double y = 1.5e-100;
    printf("\nStatus 87 before error: %X\n", _status87());
    x = y; /* create underflow and precision loss */
    printf("Status 87 after  error: %X\n", _status87());
    _clear87();
    printf("Status 87 after  clear: %X\n", _status87());
    y = x;
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.14.4 _control87, _controlfp

Header File

float.h

Category

Math Routines

Prototype

```
unsigned int _control87(unsigned int newcw, unsigned int mask);
```

```
unsigned int _controlfp(unsigned int newcw, unsigned int mask);
```

Description

Manipulates the floating-point control word.

_control87 retrieves or changes the floating-point control word.

The floating-point control word is an **unsigned int** that, bit by bit, specifies certain modes in the floating-point package; namely, the precision, infinity, and rounding modes. Changing these modes lets you mask or unmask floating-point exceptions.

_control87 matches the bits in mask to the bits in newcw. If a mask bit equals 1, the corresponding bit in newcw contains the new value for the same bit in the floating-point control word, and _control87 sets that bit in the control word to the new value.

Here is a simple illustration:

Original control word:	0100	0011	0110	0011	
mask:		1000	0001	0100	1111
newcw:		1110	1001	0000	0101
Changing bits:		1xxx	xxx1	x0xx	0101

If mask equals 0, _control87 returns the floating-point control word without altering it.

_controlfp is for Microsoft compatibility. _controlfp is identical to _control87 except that it always removes (turns off) the EM_DEMORMAL bit from the mask parameter.

Return Value

The bits in the value returned reflect the new floating-point control word. For a complete definition of the bits returned by _control87, see the header file float.h.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_control87		+		
_controlfp		+		

3.1.4.14.5 _copysign, _copysignl

Header File

float.h

Category

Math Routines

Prototype

```
double _copysign(double da, double db);
long double _copysignl(long double lda, long double ldb);
```

Description

Returns the double-precision floating point argument da, with the same sign as the double-precision floating-point argument db.

_copysignl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns the first value with the same magnitude and exponent, but with the sign of the second value. There is no error value returned.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_copysign		+		
_copysignl		+		

3.1.4.14.6 _finite, _finitel

Header File

float.h

Category

Math Routines

Prototype

```
int _finite(double d);
int _finitel(long double ld);
```

Description

Determines whether a given double-precision floating point value d is finite.

_finitel is the **long double** version; it takes a **long double** argument.

Return Value

Returns non-zero if the argument is finite, and 0 if it is not.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_finite		+		
_finitel		+		

3.1.4.14.7 _fpclass, _fpclassl

Header File

float.h

Category

Math Routines

Prototype

```
int _fpclass(double d);
int _fpclassl(long double ld);
```

Description

Returns an integer value representing the type (class) of an IEEE real for doubles. This value contains information on the floating-point class of the argument.

_fpclassl is the **long double** version; it takes a **long double** argument and returns the type (class) of an IEEE real for long doubles.

Return Value

Returns an integer value that indicates the floating-point class of its argument. The possible values, which are listed in the table below, are defined in FLOAT.H.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fpclass		+		
_fpclassl		+		

3.1.4.14.8 _fpreset

Header File

float.h

Category

Math Routines

Prototype

```
void _fpreset(void);
```

Description

Reinitializes floating-point math package.

_fpreset reinitializes the floating-point math package. This function is usually used in conjunction with system or the exec... or spawn... functions. It is also used to recover from floating-point errors before calling longjmp.

Note: If an 80x87 coprocessor is used in a program a child process (executed by the system, or by an exec... or spawn... function) might alter the parent process' floating-point state.

- If you use an 80x87 take the following precautions:
- Do not call system or an exec... or spawn... function while a floating-point expression is being evaluated.

Call _fpreset to reset the floating-point state after using system exec... or spawn... if there is any chance that the child process performed a floating-point operation with the 80x87.

Return Value

None.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.14.9 _isnan, _isnanl

Header File

float.h

Category

Classification Routines, Math Routines

Prototype

```
int _isnan(double d);
int _isnanl(long double ld);
```

Description

Tests whether a given double-precision floating-point value d is a NaN.

_isnanl is the **long double** version; it takes a **long double** argument.

Return Value

Returns a nonzero value (TRUE) if the value passed in is a NaN; otherwise it returns 0 (FALSE). The non-zero return value corresponds to either _FPCLASS_SNAN, if the NaN is of the signaling type, or _FPCLASS_QNAN, if the NaN is of the quiet type. The values for _FPCLASS_SNAN and _FPCLASS_QNAN are in float.h.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_isnan		+		
_isnanl		+		

3.1.4.14.10 _logb, _logbl

Header File

float.h

Category

Math Routines

Prototype

```
double _logb(double d);
long double _logbl(long double ld);
```

Description

Extracts the exponential value of a double-precision floating-point argument. If the argument is denormalized, it is treated as if it were normalized.

_logbl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns the unbiased exponent of the value passed in.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_logb		+		
_logbl		+		

3.1.4.14.11 _nextafter, _nextafterl

Header File

float.h

Category

Math Routines

Prototype

```
double _nextafter(double da, double db);
long double _nextafterl(long double lda, long double ldb);
```

Description

Takes two doubles (da and db) and returns the closest representable neighbor of da in the direction toward db.

If da == db, _nextafter returns da, with no exception triggered. If either da or db is a quiet NaN, then the return value is one or the other of the input NaNs.

_nextafterl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns the closest representable neighbor of the first argument in the direction toward the second argument.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_nextafter		+		
_nextafterl		+		

3.1.4.14.12 _scalb, _scalbl

Header File

float.h

Category

Math Routines

Prototype

```
double _scalb(double d, long exp);
long double _scalbl(long double ld, long exp);
```

Description

Scales the argument d by a power of 2.

_scalbl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

Returns an exponential value if successful. On overflow (depending on the sign of the argument), the function returns +/– HUGE_VAL; the ERRNO variable is set to ERANGE.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_scalb		+		
_scalbl		+		

3.1.4.14.13 _status87, _statusfp

Header File

float.h

Category

Math Routines

Prototype

```
unsigned int _status87(void);
unsigned int _statusfp(void);
```

Description

Gets floating-point status.

_status87 gets the floating-point status word, which is a combination of the 80x87 status word and other conditions detected by

the 80x87 exception handler.

_statusfp is identical to _status87 and is for Microsoft compatibility.

Return Value

The bits in the return value give the floating-point status. See float.h for a complete definition of the bits returned by _status87 and _status87.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_status87		+		
_statusfp		+		

3.1.4.15 io.h

The following functions, macros, and classes are provided in `io.h`:

Topics

Name	Description
_findclose (see page 873)	**Header File** io.h **Category** Directory Control Routines **Prototype** `int _findclose(long handle);` **Description** Closes the specified search handle and releases associated resources associated with previous calls to findfirst/findnext. The handle parameter is the search handle returned by the previous call to _findfirst. This function is provided for Microsoft compatibility. **Return Value** On success, returns 0. Otherwise, returns –1 and sets errno to: ENOENTFile specification that could not be matched **Portability**

_findfirst, __wfindfirst (see page 874)	**Header File**
	io.h, wchar.h
	Category
	Directory Control Routines
	Prototype
	`long _findfirst(char *filter, struct _finddata_t *fileinfo);`
	`long __wfindfirst(wchar_t *filter, struct _wfinddata_t *fileinfo);`
	Description
	Begins a search of a disk directory to find information about the first instance of a filename that matches a specified filter. The filter parameter is a string that specifies which files to return. Wildcards may be used in the filter. The fileinfo parameter is the file information buffer. If a matching file is found, the fileinfo structure is filled with the file-directory information.
	These functions are provided for Microsoft compatibility.
	Return Value
	On success, returns a unique... more (see page 874)

_findfirsti64, _wfindfirsti64 (see page 875)	**Header File**
	io.h, wchar.h
	Category
	Directory Control Routines
	Prototype
	`long _findfirsti64(char *filter, struct _finddatai64_t *fileinfo);`
	`long _wfindfirsti64(wchar_t *filter, struct _wfinddatai64_t *fileinfo);`
	Description
	Begins a search of a disk directory to find information about the first instance of a filename that matches a specified filter. The filter parameter is a string that specifies which files to return. Wildcards may be used in the filter. The fileinfo parameter is the file information buffer. If a matching file is found, the fileinfo structure is filled with the file-directory information.
	These i64 versions are for 64 bit filesize use and are provided for Microsoft... more (see page 875)

_findnext, __wfindnext (see page 875)	**Header File**
	io.h, wchar.h
	Category
	Directory Control Routines
	Prototype
	`long _findnext(long handle, struct _finddata_t *fileinfo);`
	`long __wfindnext(long handle, struct _wfinddata_t *fileinfo);`
	Description
	Finds subsequent files, if any, that match the filter argument in a previous call to _findfirst/__wfindfirst. Then, _findnext/__wfindnext updates the fileinfo structure with the necessary information for continuing the search. One file name for each call to _tfindnext is returned until no more files are found in the directory matching the pathname (filter).
	The handle parameter is the search handle returned by a previous call to _findfirst. The fileinfo parameter is the file information buffer.
	These functions are... more (see page 875)

_findnexti64, _wfindnexti64 (see page 876)	**Header File**
	io.h, wchar.h
	Category
	Directory Control Routines
	Prototype
	`long _findnexti64(long handle, struct _finddatai64_t *fileinfo);`
	`__int64 _wfindnexti64(long handle, struct _wfinddata_t *fileinfo);`
	Description
	Finds subsequent files, if any, that match the filter argument in a previous call to _findfirsti64/_wfindfirsti64. Then, _findnexti64/_wfindnexti64 updates the fileinfo structure with the necessary information for continuing the search. One file name for each call to _tfindnext is returned until no more files are found in the directory matching the pathname (filter).
	The handle parameter is the search handle returned by a previous call to _findfirst. The fileinfo parameter is the file information buffer.
	These i64 versions... more (see page 876)

_get_osfhandle (see page 877)	**Header File** io.h **Category** Input/output Routines **Prototype** `long _get_osfhandle(int filehandle);` **Description** Associates file handles. The _get_osfhandle function associates an operating system file handle with an existing runtime file handle. The variable filehandle is the file handle of your program. **Return value** On success, _get_osfhandle returns an operating system file handle corresponding to the variable filehandle. On error, the function returns -1 and sets the global variable errno to

_open_osfhandle (see page 879)	**Header File** io.h **Category** Input/output Routines **Prototype** `int _open_osfhandle(long osfhandle, int flags);` **Description** Associates file handles. The _open_osfhandle function allocates a runtime file handle and sets it to point to the operating system file handle specified by osfhandle. The value flags is a bitwise OR combination of one or more of the following manifest constants (defined in fcntl.h):

_rtl_chmod, _wrtl_chmod (see page 880)	**Header File**
	io.h
	Category
	Input/output Routines
	Prototype
	`int _rtl_chmod(const char *path, int func [, int attrib]);`
	`int _wrtl_chmod(const wchar_t *path, int func, ... );`
	Description
	Gets or sets file attributes.
	Note: The _rtl_chmod function replaces _chmod
	which is obsolete_rtl_chmod can either fetch or set file attributes. If func is 0, _rtl_chmod returns the current attributes for the file. If func is 1, the attribute is set to attrib.
	attrib can be one of the following symbolic constants (defined in dos.h):

_rtl_close (see page 881)	**Header File** io.h **Category** Input/output Routines **Prototype** `int _rtl_close(int handle);` **Description** Closes a file. **Note:** This function replaces _close which is obsolete The _rtl_close function closes the file associated with handle, a file handle obtained from a call to creat, creatnew, creattemp, dup, dup2, open, _rtl_creat, or _rtl_open. It does not write a Ctrl-Z character at the end of the file. If you want to terminate the file with a Ctrl-Z, you must explicitly output one. **Return Value** On success, _rtl_close returns 0. On error (if it fails because handle is not the handle of a valid, open file), _rtl_close... more (see page 881)

_rtl_creat, _wrtl_creat (see page 882)	**Header File**
	io.h
	Category
	Input/output Routines
	Prototype
	`int _rtl_creat(const char *path, int attrib);`
	`int _wrtl_creat(const wchar_t *path, int attrib);`
	Description
	Creates a new file or overwrites an existing one.
	Note: The _rtl_creat function replaces _creat,
	which is obsolete_rtl_creat opens the file specified by path. The file is always opened in binary mode. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.
	If the file already exists its size is reset to 0. (This is essentially the same as deleting the file and creating a new... more (see page 882)

_rtl_open, _wrtl_open (see page 883)	**Header File**
	io.h
	Category
	Input/output Routines
	Prototype
	`int _rtl_open(const char *filename, int oflags);`
	`int _wrtl_open(const wchar_t *path, int oflags);`
	Description
	Opens a file for reading or writing.
	Note: The _rtl_open function replaces _open which is obsolete.
	_rtl_open opens the file specified by filename, then prepares it for reading or writing, as determined by the value of oflags. The file is always opened in binary mode.
	oflags uses the flags from the following two lists. Only one flag from List 1 can be used (and one must be used) and the flags in List 2 can be used in any logical... more (see page 883)

_rtl_read (see page 885)	**Header File** io.h **Category** Input/output Routines **Prototype** `int _rtl_read(int handle, void *buf, unsigned len);` **Description** Reads from file. **Note:** This function replaces _read which is obsolete. This function reads len bytes from the file associated with handle into the buffer pointed to by buf. When a file is opened in text mode, _rtl_read does not remove carriage returns. The argument handle is a file handle obtained from a creat, open, dup, or dup2 call. On disk files, _rtl_read begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes... more (see page 885)
_rtl_write (see page 886)	**Header File** io.h **Category** Input/output Routines **Prototype** `int _rtl_write(int handle, void *buf, unsigned len);` **Description** Writes to a file. **Note:** This function replaces _write which is obsolete. _rtl_write attempts to write len bytes from the buffer pointed to by buf to the file associated with handle. The maximum number of bytes that _rtl_write can write is UINT_MAX -1 (because UINT_MAX is the same as -1), which is the error return indicator for _rtl_write. UINT_MAX is defined in limits.h. _rtl_write does not translate a linefeed character (LF) to a CR/LF pair because all its files are binary files. If the number... more (see page 886)

access, _waccess (see page 887)	**Header File**
	io.h
	Category
	Input/output Routines
	Prototype
	`int access(const char *filename, int amode);`
	`int _waccess(const wchar_t *filename, int amode);`
	Description
	Determines accessibility of a file.
	access checks the file named by filename to determine if it exists, and whether it can be read, written to, or executed.
	The list of amode values is as follows:

chmod, _wchmod (see page 889)	**Header File**
	io.h
	Category
	Input/output Routines
	Prototype
	`int chmod(const char *path, int amode);`
	`int _wchmod(const wchar_t *path, int amode);`
	Description
	Changes file access mode.
	chmod sets the file-access permissions of the file given by path according to the mask given by amode. path points to a string.
	amode can contain one or both of the symbolic constants S_IWRITE and S_IREAD (defined in sys\stat.h).

chsize (see page 890)	**Header File** io.h **Category** Input/output Routines **Prototype** `int chsize(int handle, long size);` **Description** Changes the file size. chsize changes the size of the file associated with handle. It can truncate or extend the file, depending on the value of size compared to the file's original size. The mode in which you open the file must allow writing. If chsize extends the file, it will append null characters (\0). If it truncates the file, all data beyond the new end-of-file indicator is lost. **Return Value** On success, chsize returns 0. On failure, it returns -1 and the global variable errno is... more (see page 890)
close (see page 891)	**Header File** io.h **Category** Input/output Routines **Prototype** `int close(int handle);` **Description** Closes a file. The close function closes the file associated with handle, a file handle obtained from a call to creat, creatnew, creattemp, dup, dup2, open, _rtl_creat, or _rtl_open. It does not write a Ctrl-Z character at the end of the file. If you want to terminate the file with a Ctrl-Z, you must explicitly output one. **Return Value** Upon successful completion, close returns 0. On error (if it fails because handle is not the handle of a valid, open file), close returns a value of -1 and the... more (see page 891)

_creat, _wcreat (see page 892)	**Header File**
	io.h
	Category
	Input/output Routines
	Prototype
	`int creat(const char *path, int amode);`
	`int _wcreat(const wchar_t *path, int amode);`
	Description
	Creates a new file or overwrites an existing one.
	Note: Remember that a backslash in a path requires '\\'.
	creat creates a new file or prepares to rewrite an existing file given by path. amode applies only to newly created files.
	A file created with creat is always created in the translation mode specified by the global variable _fmode (O_TEXT or O_BINARY).
	If the file exists and the write attribute is set, creat truncates the file to a length of... more (see page 892)

creatnew (see page 893)	**Header File** io.h **Category** Input/output Routines **Prototype** `int creatnew(const char *path, int mode);` **Description** Creates a new file. creatnew is identical to _rtl_creat with one exception: If the file exists, creatnew returns an error and leaves the file untouched. The mode argument to creatnew can be zero or an OR-combination of any one of the following constants (defined in dos.h):
creattemp (see page 894)	**Header File** io.h **Category** Input/output Routines **Prototype** `int creattemp(char *path, int attrib);` **Description** Creates a unique file in the directory associated with the path name. A file created with creattemp is always created in the translation mode specified by the global variable _fmode (O_TEXT or O_BINARY). path is a path name ending with a backslash (\). A unique file name is selected in the directory given by path. The newly created file name is stored in the path string supplied. path should be long enough to hold the resulting file name. The file is not automatically deleted when the program... more (see page 894)
dup (see page 895)	**Header File** io.h **Category** Input/output Routines **Prototype** `int dup(int handle);` **Description** Duplicates a file handle. • dup creates a new file handle that has the following in common with the original file handle: • Same open file or device • Same file pointer (that is, changing the file pointer of one changes the other) • Same access mode (read, write, read/write) handle is a file handle obtained from a call to creat, open, dup, dup2, _rtl_creat, or _rtl_open. **Return Value** Upon successful completion, dup returns the new file handle, a nonnegative integer; otherwise, dup returns -1. In the event of error, the global variable... more (see page 895)

dup2 (see page 896)	**Header File** io.h **Category** Input/output Routines **Prototype** `int dup2(int oldhandle, int newhandle);` **Description** Duplicates a file handle (oldhandle) onto an existing file handle (newhandle). • dup2 creates a new file handle that has the following in common with the original file handle: • Same open file or device • Same file pointer (that is, changing the file pointer of one changes the other) • Same access mode (read, write, read/write) dup2 creates a new handle with the value of newhandle. If the file associated with newhandle is open when dup2 is called, the file is closed. newhandle and oldhandle are file handles obtained from... more (see page 896)
eof (see page 898)	**Header File** io.h **Category** Input/output Routines **Prototype** `int eof(int handle);` **Description** Checks for end-of-file. eof determines whether the file associated with handle has reached end-of-file. **Return Value** If the current position is end-of-file, eof returns the value 1; otherwise, it returns 0. A return value of -1 indicates an error; the global variable errno is set to
filelength (see page 899)	**Header File** io.h **Category** Input/output Routines **Prototype** `long filelength(int handle);` **Description** Gets file size in bytes. filelength returns the length (in bytes) of the file associated with handle. **Return Value** On success filelength returns a **long** value the file length in bytes. On error it returns -1 and the global variable errno is set to

getftime, setftime (see page 899)	**Header File**
	io.h
	Category
	Input/output Routines
	Prototype
	`int getftime(int handle, struct ftime *ftimep);`
	`int setftime(int handle, struct ftime *ftimep);`
	Description
	Gets and sets the file date and time.
	getftime retrieves the file time and date for the disk file associated with the open handle. The ftime structure pointed to by ftimep is filled in with the file's time and date.
	setftime sets the file date and time of the disk file associated with the open handle to the date and time in the ftime structure pointed to by ftimep. The file must not be written to after the setftime call... more (see page 899)

isatty (see page 901)	**Header File** io.h **Category** Input/output Routines **Prototype** `int isatty(int handle);` **Description** Checks for device type. isatty determines whether handle is associated with any one of the following character devices: • a terminal • a console • a printer • a serial port **Return Value** If the device is one of the four character devices listed above, isatty returns a nonzero integer. If it is not such a device, isatty returns 0. **Example**
lock (see page 902)	**Header File** io.h **Category** Input/output Routines **Prototype** `int lock(int handle, long offset, long length);` **Description** Sets file-sharing locks. lock provides an interface to the operating system file-sharing mechanism. A lock can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error. **Return Value** lock returns 0 on success. On error, lock returns -1 and sets the global variable errno to
locking (see page 903)	**Header File** io.h, sys\locking.h **Category** Input/output Routines **Prototype** `int locking(int handle, int cmd, long length);` **Description** Sets or resets file-sharing locks. locking provides an interface to the operating system file-sharing mechanism. The file to be locked or unlocked is the open file specified by handle. The region to be locked or unlocked starts at the current file position, and is length bytes long. Locks can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with... more (see page 903)
lseek (see page 904)	**Header File** io.h **Category** Input/output Routines **Prototype** `long lseek(int handle, long offset, int fromwhere);` **Description** Moves file pointer. lseek sets the file pointer associated with handle to a new position offset bytes beyond the file location given by fromwhere. fromwhere must be one of the following symbolic constants (defined in io.h):

read (see page 905)	**Header File** io.h **Category** Input/output Routines **Prototype** `int read(int handle, void *buf, unsigned len);` **Description** Reads from file. read attempts to read len bytes from the file associated with handle into the buffer pointed to by buf. For a file opened in text mode, read removes carriage returns and reports end-of-file when it reaches a Ctrl-Z. The file handle handle is obtained from a creat, open, dup, or dup2 call. On disk files, read begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the... more (see page 905)
setmode (see page 906)	**Header File** io.h **Category** Input/output Routines **Prototype** `int setmode(int handle, int amode);` **Description** Sets mode of an open file. setmode sets the mode of the open file associated with handle to either binary or text. The argument amode must have a value of either O_BINARY or O_TEXT, never both. (These symbolic constants are defined in fcntl.h.) **Return Value** setmode returns the previous translation mode if successful. On error it returns -1 and sets the global variable errno to
tell (see page 907)	**Header File** io.h **Category** Input/output Routines **Prototype** `long tell(int handle);` **Description** Gets the current position of a file pointer. tell gets the current position of the file pointer associated with handle and expresses it as the number of bytes from the beginning of the file. **Return Value** tell returns the current file pointer position. A return of -1 (**long**) indicates an error, and the global variable errno is set to
umask (see page 908)	**Header File** io.h, sys\stat.h **Category** Input/output Routines **Prototype** `unsigned umask(unsigned mode);` **Description** Sets file read/write permission mask. The umask function sets the access permission mask used by open and creat. Bits that are set in mode will be cleared in the access permission of files subsequently created by open and creat. The mode can have one of the following values, defined in sys\stat.h:

unlock (see page 909)	**Header File** io.h **Category** Input/output Routines **Prototype** `int unlock(int handle, long offset, long length);` **Description** Releases file-sharing locks. unlock provides an interface to the operating system file-sharing mechanism. unlock removes a lock previously placed with a call to lock. To avoid error, all locks must be removed before a file is closed. A program must release all locks before completing. **Return Value** On success, unlock returns 0 O error, it returns -1. **Example**
vsscanf (see page 910)	**Header File** io.h **Category** Memory and String Manipulation Routines **Prototype** `int vsscanf(const char *buffer, const char *format, va_list arglist);` **Description** Scans and formats input from a stream. The v...scanf functions are known as alternate entry points for the ...scanf functions. They behave exactly like their ...scanf counterparts, but they accept a pointer to a list of arguments instead of an argument list. **Note:** For details on format specifiers, see Scanf Format Specifiers. vsscanf scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specifier passed to... more (see page 910)
write (see page 911)	**Header File** io.h **Category** Input/output Routines **Prototype** `int write(int handle, void *buf, unsigned len);` **Description** Writes to a file. write writes a buffer of data to the file or device named by the given handle. handle is a file handle obtained from a creat, open, dup, or dup2 call. This function attempts to write len bytes from the buffer pointed to by buf to the file associated with handle. Except when write is used to write to a text file, the number of bytes written to the file will be no more than the number requested. The maximum number of... more (see page 911)

3.1.4.15.1 _findclose

Header File

io.h

Category

Directory Control Routines

Prototype

```
int _findclose(long handle);
```

Description

Closes the specified search handle and releases associated resources associated with previous calls to findfirst/findnext. The handle parameter is the search handle returned by the previous call to _findfirst.

This function is provided for Microsoft compatibility.

Return Value

On success, returns 0.

Otherwise, returns –1 and sets errno to:

ENOENTFile specification that could not be matched

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findclose		+		

See Also

_findfirst (see page 874)

_findnext (see page 875)

3.1.4.15.2 _findfirst, __wfindfirst

Header File

io.h, wchar.h

Category

Directory Control Routines

Prototype

```
long _findfirst(char *filter, struct _finddata_t *fileinfo);
long __wfindfirst(wchar_t *filter, struct _wfinddata_t *fileinfo);
```

Description

Begins a search of a disk directory to find information about the first instance of a filename that matches a specified filter. The filter parameter is a string that specifies which files to return. Wildcards may be used in the filter. The fileinfo parameter is the file information buffer. If a matching file is found, the fileinfo structure is filled with the file-directory information.

These functions are provided for Microsoft compatibility.

Return Value

On success, returns a unique search handle to a file or group of files matching the filter specification.

Otherwise, returns –1 and sets errno to one of the following values:

ENOENTPath or file name not found

EINVALInvalid filename specification

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findfirst		+		
__wfindfirst		NT only		

See Also

_findclose (see page 873)

_findnext (see page 875)

3.1.4.15.3 _findfirsti64, _wfindfirsti64

Header File

io.h, wchar.h

Category

Directory Control Routines

Prototype

```
long _findfirsti64(char *filter, struct _finddatai64_t *fileinfo);
long _wfindfirsti64(wchar_t *filter, struct _wfinddatai64_t *fileinfo);
```

Description

Begins a search of a disk directory to find information about the first instance of a filename that matches a specified filter. The filter parameter is a string that specifies which files to return. Wildcards may be used in the filter. The fileinfo parameter is the file information buffer. If a matching file is found, the fileinfo structure is filled with the file-directory information.

These i64 versions are for 64 bit filesize use and are provided for Microsoft compatibility.

Return Value

On success, returns a unique search handle identifying the file or group of files matching the filter specification.

Otherwise, returns –1 and sets errno to one of the following values:

ENOENTFile specification that could not be matched

EINVALInvalid filename specification

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findfirsti64		+		
_wfindfirsti64		NT only		

3.1.4.15.4 _findnext, __wfindnext

Header File

io.h, wchar.h

Category

Directory Control Routines

Prototype

```
long _findnext(long handle, struct _finddata_t *fileinfo);
long __wfindnext(long handle, struct _wfinddata_t *fileinfo);
```

Description

Finds subsequent files, if any, that match the filter argument in a previous call to _findfirst/__wfindfirst. Then, _findnext/__wfindnext updates the fileinfo structure with the necessary information for continuing the search. One file name for each call to _tfindnext is returned until no more files are found in the directory matching the pathname (filter).

The handle parameter is the search handle returned by a previous call to _findfirst. The fileinfo parameter is the file information buffer.

These functions are provided for Microsoft compatibility.

Return Value

On success, returns 0.

Otherwise, returns –1 and sets errno to:

ENOENTFile specification that could not be matched

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findnext		+		
__wfindnext		NT only		

See Also

_findclose (see page 873)

_findfirst (see page 874)

3.1.4.15.5 _findnexti64, _wfindnexti64

Header File

io.h, wchar.h

Category

Directory Control Routines

Prototype

```
long _findnexti64(long handle, struct _finddatai64_t *fileinfo);
__int64 _wfindnexti64(long handle, struct _wfinddata_t *fileinfo);
```

Description

Finds subsequent files, if any, that match the filter argument in a previous call to _findfirsti64/_wfindfirsti64. Then, _findnexti64/_wfindnexti64 updates the fileinfo structure with the necessary information for continuing the search. One file name for each call to _tfindnext is returned until no more files are found in the directory matching the pathname (filter).

The handle parameter is the search handle returned by a previous call to _findfirst. The fileinfo parameter is the file information buffer.

These i64 versions are for 64 bit filesize use and are provided for Microsoft compatibility.

Return Value

On success, returns 0.

Otherwise, returns –1 and sets errno to:

ENOENTFile specification that could not be matched

Portability

	POSIX	Win32	ANSI C	ANSI C++
_findnexti64		+		
_wfindnexti64		NT only		

3.1.4.15.6 _get_osfhandle

Header File

io.h

Category

Input/output Routines

Prototype

```
long _get_osfhandle(int filehandle);
```

Description

Associates file handles.

The _get_osfhandle function associates an operating system file handle with an existing runtime file handle. The variable filehandle is the file handle of your program.

Return value

On success, _get_osfhandle returns an operating system file handle corresponding to the variable filehandle.

On error, the function returns -1 and sets the global variable errno to

EBADF	an invalid file handle

Example

```
#include <windows.h>
#include <fcntl.h>
#include <stdio.h>
#include <io.h>
//Example for _get_osfhandle() and _open_osfhandle()
BOOL InitApplication(HINSTANCE hInstance);
HWND InitInstance(HINSTANCE hInstance, int nCmdShow);
LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam);
Example_get_osfhandle(HWND hWnd);
#pragma argsused
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
MSG msg;      // message
  if (!InitApplication(hInstance))  // Initialize shared things
   return (FALSE); // Exits if unable to initialize
  /* Perform initializations that apply to a specific instance */
  if (!(InitInstance(hInstance, nCmdShow)))
   return (FALSE);
  /* Acquire and dispatch messages until a WM_QUIT message is received. */
```

```
    while (GetMessage(&msg, // message structure
     NULL, // handle of window receiving the message
     NULL, // lowest message to examine
     NULL))  // highest message to examine
    {
    TranslateMessage(&msg); // Translates virtual key codes
    DispatchMessage(&msg);  // Dispatches message to window
    }
    return (msg.wParam);  // Returns the value from PostQuitMessage
  }
  BOOL InitApplication(HINSTANCE hInstance)
  {
    WNDCLASS  wc;
    // Fill in window class structure with parameters that describe the
    // main window.
    wc.style = CS_HREDRAW | CS_VREDRAW; // Class style(s).
    wc.lpfnWndProc = (long (FAR PASCAL*)(void *,unsigned int,unsigned int, long ))MainWndProc;
  // Function to retrieve messages for
                  // windows of this class.
    wc.cbClsExtra = 0;  // No per-class extra data.
    wc.cbWndExtra = 0;  // No per-window extra data.
    wc.hInstance = hInstance; // Application that owns the class.
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;  // Name of menu resource in .RC file.
    wc.lpszClassName = "Example"; // Name used in call to CreateWindow.
    /* Register the window class and return success/failure code. */
    return (RegisterClass(&wc));
  }
  HWND InitInstance(HINSTANCE hInstance, int nCmdShow)
  {
    HWND  hWnd; // Main window handle.
    /* Create a main window for this application instance.  */
    hWnd = CreateWindow(
     "Example",                  // See RegisterClass() call.
     "Example _get_osfhandle _open_osfhandle (32 bit)",  // Text for window title bar.
     WS_OVERLAPPEDWINDOW,        // Window style.
     CW_USEDEFAULT,          // Default horizontal position.
     CW_USEDEFAULT,           // Default vertical position.
     CW_USEDEFAULT,             // Default width.
     CW_USEDEFAULT,         // Default height.
     NULL,                   // Overlapped windows have no parent.
     NULL,                   // Use the window class menu.
     hInstance,               // This instance owns this window.
     NULL                    // Pointer not needed.
    );
    /* If window could not be created, return "failure" */
    if (!hWnd)
     return (FALSE);
    /* Make the window visible; update its client area; and return "success" */
    ShowWindow(hWnd, nCmdShow); // Show the window
    UpdateWindow(hWnd);     // Sends WM_PAINT message
    return (hWnd);        // Returns the value from PostQuitMessage
  }
  #pragma argsused
  LRESULT FAR PASCAL _export MainWndProc(HWND hWnd, UINT message,
              WPARAM wParam, LPARAM lParam)
  {
    switch (message)
    {
      case WM_CREATE:
      {
        Example_get_osfhandle(hWnd);
        return NULL;
      }
```

```
    case WM_QUIT:
    case WM_DESTROY:  // message: window being destroyed
      PostQuitMessage(0);
    break;
    default:      // Passes it on if unprocessed
      return (DefWindowProc(hWnd, message, wParam, lParam));
  }
}
Example_get_osfhandle(HWND hWnd)
{
    long osfHandle;
    char str[128];
    int fHandle = open("file1.c", O_CREAT|O_TEXT);
    if(fHandle != -1)
    {
      osfHandle = _get_osfhandle(fHandle);
      sprintf(str, "file handle = %lx OS file handle = %lx", fHandle, osfHandle);
      MessageBox(hWnd,str,"_get_osfhandle",MB_OK|MB_ICONINFORMATION);
      close(fHandle);
      fHandle = _open_osfhandle(osfHandle, O_TEXT );
      sprintf(str, "file handle = %lx OS file handle = %lx", fHandle, osfHandle);
      MessageBox(hWnd,str,"_open_osfhandle",MB_OK|MB_ICONINFORMATION);
         close(fHandle);
    }
    else
      MessageBox(hWnd,"File Open Error","WARNING",MB_OK|MB_ICONSTOP);
return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.7 _open_osfhandle

Header File

io.h

Category

Input/output Routines

Prototype

```
int _open_osfhandle(long osfhandle, int flags);
```

Description

Associates file handles.

The _open_osfhandle function allocates a runtime file handle and sets it to point to the operating system file handle specified by osfhandle.

The value flags is a bitwise OR combination of one or more of the following manifest constants (defined in fcntl.h):

O_APPEND	Repositions the file pointer to the end of the file before every write operation.
O_RDONLY	Opens the file for reading only.
O_TEXT	Opens the file in text (translated) mode.

Return Value

On success, _open_osfhandle returns a C runtime file handle. Otherwise, it returns -1.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.8 _rtl_chmod, _wrtl_chmod

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_chmod(const char *path, int func [, int attrib]);
int _wrtl_chmod(const wchar_t *path, int func, ... );
```

Description

Gets or sets file attributes.

Note: The _rtl_chmod function replaces _chmod

which is obsolete_rtl_chmod can either fetch or set file attributes. If func is 0, _rtl_chmod returns the current attributes for the file. If func is 1, the attribute is set to attrib.

attrib can be one of the following symbolic constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file
FA_LABEL	Volume label
FA_DIREC	Directory
FA_ARCH	Archive

Return Value

On success, _rtl_chmod returns the file attribute word.

On error, it returns a value of -1 and sets the global variable errno to one of the following values:

ENOENT	Path or filename not found
EACCES	Permission denied

Example

```
#include <errno.h>
#include <stdio.h>
#include <dir.h>
#include <io.h>
```

```
int get_file_attrib(char *filename);
int main(void)
{
  char filename[128];
  int attrib;
  printf("Enter a filename:");
  scanf("%s", filename);
  attrib = get_file_attrib(filename);
  if (attrib == -1)
     switch(errno)
     {
        case ENOENT : printf("Path or file not found.\n");
                      break;
        case EACCES : printf("Permission denied.\n");
                      break;
        default:      printf("Error number: %d", errno);
                      break;
     }
  else
     {
        if (attrib & FA_RDONLY)
           printf("%s is read-only.\n", filename);
        if (attrib & FA_HIDDEN)
           printf("%s is hidden.\n", filename);
        if (attrib & FA_SYSTEM)
           printf("%s is a system file.\n", filename);
        if (attrib & FA_DIREC)
           printf("%s is a directory.\n", filename);
        if (attrib & FA_ARCH)
           printf("%s is an archive file.\n", filename);
     }
  return 0;
}
/* returns the attributes of a DOS file */
int get_file_attrib(char *filename)
{
   return(_rtl_chmod(filename, 0));
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rtl_chmod		+		
_wrtl_chmod		NT only		

3.1.4.15.9 _rtl_close

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_close(int handle);
```

Description

Closes a file.

Note: This function replaces _close which is obsolete

The _rtl_close function closes the file associated with handle, a file handle obtained from a call to creat, creatnew, creattemp, dup, dup2, open, _rtl_creat, or _rtl_open.

It does not write a Ctrl-Z character at the end of the file. If you want to terminate the file with a Ctrl-Z, you must explicitly output one.

Return Value

On success, _rtl_close returns 0.

On error (if it fails because handle is not the handle of a valid, open file), _rtl_close returns a value of -1 and the global variable errno is set to

EBADF	Bad file number

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "Hello world";
   if ((handle = _rtl_open("TEST.$$$", O_RDWR)) == -1)
   {
      perror("Error:");
      return 1;
   }
   _rtl_write(handle, msg, strlen(msg));
   _rtl_close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.10 _rtl_creat, _wrtl_creat

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_creat(const char *path, int attrib);
```

```
int _wrtl_creat(const wchar_t *path, int attrib);
```

Description

Creates a new file or overwrites an existing one.

Note: The _rtl_creat function replaces _creat,

which is obsolete_rtl_creat opens the file specified by path. The file is always opened in binary mode. Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

If the file already exists its size is reset to 0. (This is essentially the same as deleting the file and creating a new file with the same name.)

The attrib argument is an OR'ed combination of one or more of the following constants (defined in dos.h):

FA_RDONLY	Read-only attribute
FA_HIDDEN	Hidden file
FA_SYSTEM	System file

Return Value

On success, _rtl_creat returns the new file handle (a non-negative integer).

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCESPermission denied

EMFILEToo many open files

ENOENTPath or file name not found

Example

```
#include <string.h>
#include <stdio.h>
#include <io.h>
int main() {
  unsigned count;
  int handle;
  char buf[11] = "0123456789";
  /* Create a 10-byte file using _rtl_creat. */
  if ((handle = _rtl_creat("DUMMY2.FIL", 0)) < 0) {
    perror("Unable to _rtl_create DUMMY2.FIL");
    return 1;
  }
  if (_rtl_write(handle, buf, strlen(buf)) < 0) {
    perror("Unable to _rtl_write to DUMMY2.FIL");
    return 1;
  }
  _rtl_close(handle);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rtl_crea		+		
_wrtl_creat		NT only		

3.1.4.15.11 _rtl_open, _wrtl_open

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_open(const char *filename, int oflags);
```

```
int _wrtl_open(const wchar_t *path, int oflags);
```

Description

Opens a file for reading or writing.

Note: The _rtl_open function replaces _open which is obsolete.

_rtl_open opens the file specified by filename, then prepares it for reading or writing, as determined by the value of oflags. The file is always opened in binary mode.

oflags uses the flags from the following two lists. Only one flag from List 1 can be used (and one must be used) and the flags in List 2 can be used in any logical combination.

O_RDONLY	Open for reading.
O_WRONLY	Open for writing.
O_RDWR	Open for reading and writing.

The following additional values can be included in oflags (using an OR operation):

O_NOINHERIT	The file is not passed to child programs.
SH_COMPAT	Allow other opens with SH_COMPAT. All other openings of a file with the SH_COMPAT flag must be opened using SH_COMPAT flag. You can request a file open that uses SH_COMPAT logically OR'ed with some other flag (for example, SH_COMPAT | SH_DENWR is allowed). The call will fail if the file has already been opened in any other shared mode.
SH_DENYRW	Only the current handle can have access to the file.
SH_DENWR	Allow only reads from any other open to the file.
SH_DENYRD	Allow only writes from any other open to the file.
SH_DENYNO	Allow other shared opens to the file, but not other SH_COMPAT opens.

Note: These symbolic constants are defined in fcntl.h and share.h.

Only one of the SH_DENYxx values can be included in a single _rtl_open routine. These file-sharing attributes are in addition to any locking performed on the files.

The maximum number of simultaneously open files is defined by HANDLE_MAX.

Return Value

On success:_rtl_open returns a non-negative integer (the file handle). The file pointer, which marks the current position in the file, is set to the beginning of the file.

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EINVACC	Invalid access code
EMFILE	Too many open files
ENOENT	Path or file not found

Example

```
#include <string.h>
#include <stdio.h>
```

```
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "Hello world";
   if ((handle = _rtl_open("TEST.$$$", O_RDWR)) == -1)
   {
      perror("Error:");
      return 1;
   }
   _rtl_write(handle, msg, strlen(msg));
   _rtl_close(handle);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rtl_open		+		
_wrtl_open		NT only		

3.1.4.15.12 _rtl_read

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_read(int handle, void *buf, unsigned len);
```

Description

Reads from file.

Note: This function replaces _read which is obsolete.

This function reads len bytes from the file associated with handle into the buffer pointed to by buf. When a file is opened in text mode, _rtl_read does not remove carriage returns.

The argument handle is a file handle obtained from a creat, open, dup, or dup2 call.

On disk files, _rtl_read begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes it can read is UINT_MAX -1 (because UINT_MAX is the same as -1, the error return indicator). UINT_MAX is defined in limits.h.

Return Value

On success, _rtl_read returns either

- a positive integer, indicating the number of bytes placed in the buffer
- zero, indicating end-of-file

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Example

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>
int main(void)
{
   void *buf;
   int handle, bytes;
   buf = malloc(10);
/*
Looks for a file in the current directory named TEST.$$$ and attempts to read 10 bytes from
it. To use this example you should create the file TEST.$$$
 */
   if ((handle =
       open("TEST.$$$", O_RDONLY | O_BINARY, S_IWRITE | S_IREAD)) == -1)
   {
      printf("Error Opening File\n");
      exit(1);
   }
   if ((bytes = _rtl_read(handle, buf, 10)) == -1) {
      printf("Read Failed.\n");
      exit(1);
   }
   else {
      printf("_rtl_read: %d bytes read.\n", bytes);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.13 _rtl_write

Header File

io.h

Category

Input/output Routines

Prototype

```
int _rtl_write(int handle, void *buf, unsigned len);
```

Description

Writes to a file.

Note: This function replaces _write which is obsolete.

_rtl_write attempts to write len bytes from the buffer pointed to by buf to the file associated with handle.

The maximum number of bytes that _rtl_write can write is UINT_MAX -1 (because UINT_MAX is the same as -1), which is the error return indicator for _rtl_write. UINT_MAX is defined in limits.h. _rtl_write does not translate a linefeed character (LF) to a CR/LF pair because all its files are binary files.

If the number of bytes actually written is less than that requested the condition should be considered an error and probably indicates a full disk.

For disk files, writing always proceeds from the current file pointer. On devices, bytes are directly sent to the device.

For files opened with the O_APPEND option, the file pointer is not positioned to EOF before writing the data.

Return Value

On success, _rtl_write returns number of bytes written.

On error, it returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Example

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>
int main(void)
{
   void *buf;
   int handle, bytes;
   buf = malloc(200);
/*
Create a file name TEST.$$$ in the current directory and writes 200 bytes to it. If TEST.$$$
already exists, it's overwritten.
 */
   if ((handle = open("TEST.$$$", O_CREAT | O_WRONLY | O_BINARY,
                      S_IWRITE | S_IREAD)) == -1)
   {
      printf("Error Opening File\n");
      exit(1);
   }
   if ((bytes = _rtl_write(handle, buf, 200)) == -1) {
      printf("Write Failed.\n");
      exit(1);
   }
   printf("_rtl_write: %d bytes written.\n",bytes);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.14 access, _waccess

Header File

io.h

Category

Input/output Routines

Prototype

```
int access(const char *filename, int amode);
```

```
int _waccess(const wchar_t *filename, int amode);
```

Description

Determines accessibility of a file.

access checks the file named by filename to determine if it exists, and whether it can be read, written to, or executed.

The list of amode values is as follows:

06	Check for read and write permission
04	Check for read permission
02	Check for write permission
01	Execute (ignored)
00	Check for existence of file

All existing files have read access (amode equals 04), so 00 and 04 give the same result. Similarly, amode values of 06 and 02 are equivalent because under Win32 write access implies read access.

If filename refers to a directory, access simply determines whether the directory exists.

Return Value

If the requested access is allowed, access returns 0; otherwise, it returns a value of -1, and the global variable errno is set to one of the following values:

ENOENT	Path or file name not found
EACCES	Permission denied

Example

```
 #include <stdio.h>
 #include <io.h>
 int file_exists(char *filename);
 int main(void)
 {
printf("Does NOTEXIST.FIL exist: %s\n",
 file_exists("NOTEXISTS.FIL") ? "YES" : "NO");
return 0;
 }
 int file_exists(char *filename)
 {
   return (access(filename, 0) == 0);
 }
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
access	+	+		
_waccess		NT only		

3.1.4.15.15 chmod, _wchmod

Header File

io.h

Category

Input/output Routines

Prototype

```
int chmod(const char *path, int amode);
int _wchmod(const wchar_t *path, int amode);
```

Description

Changes file access mode.

chmod sets the file-access permissions of the file given by path according to the mask given by amode. path points to a string.

amode can contain one or both of the symbolic constants S_IWRITE and S_IREAD (defined in sys\stat.h).

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD | S_IWRITE	Permission to read and write (write permission implies read permission)

Return Value

Upon successfully changing the file access mode, chmod returns 0. Otherwise, chmod returns a value of -1.

In the event of an error, the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	Path or file name not found

Example

```
#include <errno.h>
#include <stdio.h>
#include <io.h>
#include <process.h>
#include <sys\stat.h>
void main(void)
{
  char filename[64];
  struct stat stbuf;
  int amode;
  printf("Enter name of file: ");
  scanf("%s", filename);
  if (stat(filename, &stbuf) != 0)
  {
    perror("Unable to get file information");
    exit(1);
  }
  if (stbuf.st_mode & S_IWRITE)
  {
    printf("Changing to read-only\n");
    amode = S_IREAD;
  }
  else
```

```
  {
    printf("Changing to read-write\n");
    amode = S_IREAD|S_IWRITE;
  }
  if (chmod(filename, amode) != 0)
  {
    perror("Unable to change file mode");
    exit(1);
  }
  exit(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
chmod	+	+		
_wchmod		NT only		

3.1.4.15.16 chsize

Header File

io.h

Category

Input/output Routines

Prototype

```
int chsize(int handle, long size);
```

Description

Changes the file size.

chsize changes the size of the file associated with handle. It can truncate or extend the file, depending on the value of size compared to the file's original size.

The mode in which you open the file must allow writing.

If chsize extends the file, it will append null characters (\0). If it truncates the file, all data beyond the new end-of-file indicator is lost.

Return Value

On success, chsize returns 0. On failure, it returns -1 and the global variable errno is set to one of the following values:

EACCES	Permission denied
EBADF	Bad file number
ENOSPC	No space left on device

Example

```
#include <string.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char buf[11] = "0123456789";
```

```
    /* create text file containing 10 bytes */
    handle = open("DUMMY.FIL", O_CREAT);
    write(handle, buf, strlen(buf));
    /* truncate the file to 5 bytes in size */
    chsize(handle, 5);
    /* close the file */
    close(handle);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.17 close

Header File

io.h

Category

Input/output Routines

Prototype

```
int close(int handle);
```

Description

Closes a file.

The close function closes the file associated with handle, a file handle obtained from a call to creat, creatnew, creattemp, dup, dup2, open, _rtl_creat, or _rtl_open.

It does not write a Ctrl-Z character at the end of the file. If you want to terminate the file with a Ctrl-Z, you must explicitly output one.

Return Value

Upon successful completion, close returns 0.

On error (if it fails because handle is not the handle of a valid, open file), close returns a value of -1 and the global variable errno is set to

EBADF	Bad file number

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
main()
{
  int handle;
  char buf[11] = "0123456789";
  /* create a file containing 10 bytes */
  handle = open("NEW.FIL", O_CREAT);
  if (handle > -1)
  {
    write(handle, buf, strlen(buf));
```

```
    close(handle);                    /* close the file */
  }
  else
  {
    printf("Error opening file\n");
  }
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.15.18 _creat, _wcreat

Header File

io.h

Category

Input/output Routines

Prototype

```
int creat(const char *path, int amode);
```

```
int _wcreat(const wchar_t *path, int amode);
```

Description

Creates a new file or overwrites an existing one.

Note: Remember that a backslash in a path requires '\\'.

creat creates a new file or prepares to rewrite an existing file given by path. amode applies only to newly created files.

A file created with creat is always created in the translation mode specified by the global variable _fmode (O_TEXT or O_BINARY).

If the file exists and the write attribute is set, creat truncates the file to a length of 0 bytes, leaving the file attributes unchanged. If the existing file has the read-only attribute set, the creat call fails and the file remains unchanged.

The creat call examines only the S_IWRITE bit of the access-mode word amode. If that bit is 1, the file can be written to. If the bit is 0, the file is marked as read-only. All other operating system attributes are set to 0.

amode can be one of the following (defined in sys\stat.h):

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD / S_IWRITE	Permission to read and write (write permission implies read permission)

Return Value

Upon successful completion, _creat returns the new file handle, a nonnegative integer; otherwise, it returns -1.

In the event of error, the global variable errno is set to one of the following:

EACCES	Permission denied
ENOENT	Path or file name not found
EMFILE	Too many open files

Example

```
#include <sys\stat.h>
#include <string.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char buf[11] = "0123456789";
   /* change the default file mode from text to binary */
   _fmode = O_BINARY;
   /* create a binary file for reading and writing */
   handle = creat("DUMMY.FIL", S_IREAD |S_IWRITE);
   /* write 10 bytes to the file */
   write(handle, buf, strlen(buf));
   /* close the file */
   close(handle);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
creat	+	+		
_wcreat		NT only		

3.1.4.15.19 creatnew

Header File

io.h

Category

Input/output Routines

Prototype

```
int creatnew(const char *path, int mode);
```

Description

Creates a new file.

creatnew is identical to _rtl_creat with one exception: If the file exists, creatnew returns an error and leaves the file untouched.

The mode argument to creatnew can be zero or an OR-combination of any one of the following constants (defined in dos.h):

FA_HIDDEN	Hidden file
FA_RDONLY	Read-only attribute
FA_SYSTEM	System file

Return Value

Upon successful completion, creatnew returns the new file handle, a nonnegative integer; otherwise, it returns -1.

In the event of error, the global variable errno is set to one of the following values:

EACCES	Permission denied
EEXIST	File already exists
EMFILE	Too many open files
ENOENT	Path or file name not found

Example

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <io.h>
int main(void)
{
   int handle;
   char buf[11] = "0123456789";
   /* attempt to create a file that doesn't already exist */
   handle = creatnew("DUMMY.FIL", 0);
if (handle == -1)
      printf("DUMMY.FIL already exists.\n");
   else
   {
      printf("DUMMY.FIL successfully created.\n");
      write(handle, buf, strlen(buf));
      close(handle);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.20 creattemp

Header File

io.h

Category

Input/output Routines

Prototype

```
int creattemp(char *path, int attrib);
```

Description

Creates a unique file in the directory associated with the path name.

A file created with creattemp is always created in the translation mode specified by the global variable _fmode (O_TEXT or O_BINARY).

path is a path name ending with a backslash (\). A unique file name is selected in the directory given by path. The newly created file name is stored in the path string supplied. path should be long enough to hold the resulting file name. The file is not

automatically deleted when the program terminates.

The attrib argument to creattemp can be zero or an OR-combination of any one of the following constants (defined in dos.h):

FA_HIDDEN	Hidden file
FA_RDONLY	Read-only attribute
FA_SYSTEM	System file

Upon successful file creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

Return Value

Upon successful completion, the new file handle, a nonnegative integer, is returned; otherwise, -1 is returned.

In the event of error, the global variable errno is set to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found

Example

```
#include <string.h>
#include <stdio.h>
#include <io.h>
int main(void)
{
   int handle;
   char pathname[128];
   strcpy(pathname, "\\");
   /* create a unique file in the root directory */
   handle = creattemp(pathname, 0);
   printf("%s was the unique file created.\n", pathname);
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.21 dup

Header File

io.h

Category

Input/output Routines

Prototype

```
int dup(int handle);
```

Description

Duplicates a file handle.

- dup creates a new file handle that has the following in common with the original file handle:
- Same open file or device
- Same file pointer (that is, changing the file pointer of one changes the other)
- Same access mode (read, write, read/write)

handle is a file handle obtained from a call to creat, open, dup, dup2, _rtl_creat, or _rtl_open.

Return Value

Upon successful completion, dup returns the new file handle, a nonnegative integer; otherwise, dup returns -1.

In the event of error, the global variable errno is set to one of the following values:

EBADF	Bad file number
EMFILE	Too many open files

Example

```
#include <string.h>
#include <stdio.h>
#include <io.h>
void flush(FILE *stream);
int main(void)
{
  FILE *fp;
  char msg[] = "This is a test";
  /* create a file */
  fp = fopen("DUMMY.FIL", "w");
  /* write some data to the file */
  fwrite(msg, strlen(msg), 1, fp);
  printf("Press ENTER to flush DUMMY.FIL:");
  getchar();
  /* flush the data to DUMMY.FIL without closing it */
  flush(fp);
  printf("\nFile was flushed, Press ENTER to quit:");
  getchar();
  return 0;
}
void flush(FILE *stream)
{
  int duphandle;
  /* flush TC's internal buffer */
  fflush(stream);
  /* make a duplicate file handle */
  duphandle = dup(fileno(stream));
  /* close the duplicate handle to flush the DOS buffer */
  close(duphandle);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.15.22 dup2

Header File

io.h

Category

Input/output Routines

Prototype

```
int dup2(int oldhandle, int newhandle);
```

Description

Duplicates a file handle (oldhandle) onto an existing file handle (newhandle).

- dup2 creates a new file handle that has the following in common with the original file handle:
- Same open file or device
- Same file pointer (that is, changing the file pointer of one changes the other)
- Same access mode (read, write, read/write)

dup2 creates a new handle with the value of newhandle. If the file associated with newhandle is open when dup2 is called, the file is closed.

newhandle and oldhandle are file handles obtained from a creat, open, dup, or dup2 call.

Return Value

dup2 returns 0 on successful completion, -1 otherwise.

In the event of error, the global variable errno is set to one of the following values:

EBADF Bad file number

EMFILE Too many open files

Example

```
#include <sys\stat.h>
#include <string.h>
#include <fcntl.h>
#include <io.h>
#define STDOUT 1
int main(void)
{
   int nul, oldstdout;
   char msg[] = "This is a test";
   /* create a file */
   nul = open("DUMMY.FIL", O_CREAT | O_RDWR,
      S_IREAD | S_IWRITE);
   /* create a duplicate handle for standard output */
   oldstdout = dup(STDOUT);
   /*
      redirect standard output to DUMMY.FIL
      by duplicating the file handle onto
      the file handle for standard output.
   */
   dup2(nul, STDOUT);
   /* close the handle for DUMMY.FIL */
   close(nul);
   /* will be redirected into DUMMY.FIL */
   write(STDOUT, msg, strlen(msg));
   /* restore original standard output handle */
   dup2(oldstdout, STDOUT);
   /* close duplicate handle for STDOUT */
   close(oldstdout);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.15.23 eof

Header File

io.h

Category

Input/output Routines

Prototype

```
int eof(int handle);
```

Description

Checks for end-of-file.

eof determines whether the file associated with handle has reached end-of-file.

Return Value

If the current position is end-of-file, eof returns the value 1; otherwise, it returns 0. A return value of -1 indicates an error; the global variable errno is set to

EBADF	Bad file number

Example

```
#include <sys\stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "This is a test";
   char ch;
   /* create a file */
   handle = open("DUMMY.FIL",
                 O_CREAT | O_RDWR,
                 S_IREAD | S_IWRITE);
   /* write some data to the file */
   write(handle, msg, strlen(msg));
   /* seek to the beginning of the file */
   lseek(handle, 0L, SEEK_SET);
   /* reads chars from the file until it reaches EOF */
   do
   {
      read(handle, &ch, 1);
      printf("%c", ch);
   } while (!eof(handle));
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.24 filelength

Header File

io.h

Category

Input/output Routines

Prototype

```
long filelength(int handle);
```

Description

Gets file size in bytes.

filelength returns the length (in bytes) of the file associated with handle.

Return Value

On success filelength returns a **long** value the file length in bytes. On error it returns -1 and the global variable errno is set to

EBADF	Bad file number

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char buf[11] = "0123456789";
   /* create a file containing 10 bytes */
   handle = open("DUMMY.FIL", O_CREAT);
   write(handle, buf, strlen(buf));
   /* display the size of the file */
   printf("file length in bytes: %ld\n", filelength(handle));
   /* close the file */
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.25 getftime, setftime

Header File

io.h

Category

Input/output Routines

Prototype

```
int getftime(int handle, struct ftime *ftimep);
int setftime(int handle, struct ftime *ftimep);
```

Description

Gets and sets the file date and time.

getftime retrieves the file time and date for the disk file associated with the open handle. The ftime structure pointed to by ftimep is filled in with the file's time and date.

setftime sets the file date and time of the disk file associated with the open handle to the date and time in the ftime structure pointed to by ftimep. The file must not be written to after the setftime call or the changed information will be lost. The file must be open for writing; an EACCES error will occur if the file is open for read-only access.

setftime requires the file to be open for writing; an EACCES error will occur if the file is open for read-only access.

The ftime structure is defined as follows:

```
struct ftime {
unsigned ft_tsec: 5; /* two seconds */
unsigned ft_min: 6; /* minutes */
unsigned ft_hour: 5; /* hours */
unsigned ft_day: 5; /* days */
unsigned ft_month: 4; /* months */
unsigned ft_year: 7; /* year - 1980*/
};
```

Return Value

getftime and setftime return 0 on success.

In the event of an error return -1 is returned and the global variable errno is set to one of the following values:

EACCES	Permission denied
EBADF	Bad file number
EINVFNC	Invalid function number

Example

```
#include <stdio.h>
#include <io.h>
int main(void)
{
   FILE *stream;
   std::ftime ft;
   if ((stream = fopen("TEST.$$$",
        "wt")) == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }
   getftime(fileno(stream), &ft);
```

```
    printf("File time: %u:%u:%u\n",
           ft.ft_hour, ft.ft_min,
           ft.ft_tsec * 2);
    printf("File date: %u/%u/%u\n",
           ft.ft_month, ft.ft_day,
           ft.ft_year+1980);
    fclose(stream);
    return 0;
}
```

3.1.4.15.26 isatty

Header File

io.h

Category

Input/output Routines

Prototype

```
int isatty(int handle);
```

Description

Checks for device type.

isatty determines whether handle is associated with any one of the following character devices:

- a terminal
- a console
- a printer
- a serial port

Return Value

If the device is one of the four character devices listed above, isatty returns a nonzero integer. If it is not such a device, isatty returns 0.

Example

```
#include <stdio.h>
#include <io.h>
int main(void)
{
    int handle;
    handle = fileno(stdout);
    if (isatty(handle))
       printf("Handle %d is a device type\n", handle);
    else
       printf("Handle %d isn't a device type\n", handle);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.15.27 lock

Header File

io.h

Category

Input/output Routines

Prototype

```
int lock(int handle, long offset, long length);
```

Description

Sets file-sharing locks.

lock provides an interface to the operating system file-sharing mechanism.

A lock can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error.

Return Value

lock returns 0 on success. On error, lock returns -1 and sets the global variable errno to

EACCES	Locking violation

Example

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
int main(void)
{
   int handle, status;
   long length;
   handle = _sopen("c:\\autoexec.bat",
      O_RDONLY,SH_DENYNO,S_IREAD);
   if (handle < 0)
   {
      printf("_sopen failed\n");
      exit(1);
   }
   length = filelength(handle);
   status = lock(handle,0L,length/2);
   if (status == 0)
      printf("lock succeeded\n");
   else
      printf("lock failed\n");
   status = unlock(handle,0L,length/2);
   if (status == 0)
      printf("unlock succeeded\n");
   else
      printf("unlock failed\n");
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.28 locking

Header File

io.h, sys\locking.h

Category

Input/output Routines

Prototype

```
int locking(int handle, int cmd, long length);
```

Description

Sets or resets file-sharing locks.

locking provides an interface to the operating system file-sharing mechanism. The file to be locked or unlocked is the open file specified by handle. The region to be locked or unlocked starts at the current file position, and is length bytes long.

Locks can be placed on arbitrary, nonoverlapping regions of any file. A program attempting to read or write into a locked region will retry the operation three times. If all three retries fail, the call fails with an error.

The cmd specifies the action to be taken (the values are defined in sys\locking.h):

LK_LOCK	Lock the region. If the lock is unsuccessful, try once a second for 10 seconds before giving up.
LK_RLCK	Same as LK_LOCK.
LK_NBLCK	Lock the region. If the lock if unsuccessful, give up immediately.
LK_NBRLCK	Same as LK_NBLCK.
LK_UNLCK	Unlock the region, which must have been previously locked.

Return Value

On successful operations, locking returns 0. Otherwise, it returns -1, and the global variable errno is set to one of the following values:

EACCES	File already locked or unlocked
EBADF	Bad file number
EDEADLOCK	File cannot be locked after 10 retries (cmd is LK_LOCK or LK_RLCK)
EINVAL	Invalid cmd, or SHARE.EXE not loaded

Example

```
#include <io.h>
#include <fcntl.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
#include <sys\locking.h>
int main(void)
{
  int handle, status;
```

```
  long length;
  handle = sopen("c:\\autoexec.bat", O_RDONLY,SH_DENYNO);
  if (handle < 0) {
    printf("sopen failed\n");
    exit(1);
  }
  length = filelength(handle);
  status = locking(handle,LK_LOCK,length/2);
  if (status == 0)
    printf("lock succeeded\n");
  else
    perror("lock failed");
  status = locking(handle,LK_UNLCK,length/2);
  if (status == 0)
    printf("unlock succeeded\n");
  else
    perror("unlock failed");
  close(handle);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.29 lseek

Header File

io.h

Category

Input/output Routines

Prototype

```
long lseek(int handle, long offset, int fromwhere);
```

Description

Moves file pointer.

lseek sets the file pointer associated with handle to a new position offset bytes beyond the file location given by fromwhere. fromwhere must be one of the following symbolic constants (defined in io.h):

SEEK_CUR	Current file pointer position
SEEK_END	End-of-file
SEEK_SET	File beginning

Return Value

lseek returns the offset of the pointer's new position measured in bytes from the file beginning. lseek returns -1L on error, and the global variable errno is set to one of the following values:

EBADF	Bad file handle
EINVAL	Invalid argument
ESPIPE	Illegal seek on device

On devices incapable of seeking (such as terminals and printers), the return value is undefined.

Example

```
#include <sys\stat.h>
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
    int handle;
    char msg[] = "This is a test";
    char ch;
    /* create a file */
    handle = open("TEST.$$$", O_CREAT | O_RDWR, S_IREAD | S_IWRITE);
    /* write some data to the file */
    write(handle, msg, strlen(msg));
    /* seek to the beginning of the file */
    lseek(handle, 0L, SEEK_SET);
    /* reads chars from the file until we hit EOF */
    do
    {
       read(handle, &ch, 1);
       printf("%c", ch);
    }  while (!eof(handle));
    close(handle);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.15.30 read

Header File

io.h

Category

Input/output Routines

Prototype

```
int read(int handle, void *buf, unsigned len);
```

Description

Reads from file.

read attempts to read len bytes from the file associated with handle into the buffer pointed to by buf.

For a file opened in text mode, read removes carriage returns and reports end-of-file when it reaches a Ctrl-Z.

The file handle handle is obtained from a creat, open, dup, or dup2 call.

On disk files, read begins reading at the current file pointer. When the reading is complete, it increments the file pointer by the number of bytes read. On devices, the bytes are read directly from the device.

The maximum number of bytes that read can read is UINT_MAX -1, because UINT_MAX is the same as -1, the error return indicator. UINT_MAX is defined in limits.h.

Return Value

On successful completion, read returns an integer indicating the number of bytes placed in the buffer. If the file was opened in text mode, read does not count carriage returns or Ctrl-Z characters in the number of bytes read.

On end-of-file, read returns 0. On error, read returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Example

```
#include <stdio.h>
#include <io.h>
#include <alloc.h>
#include <fcntl.h>
#include <process.h>
#include <sys\stat.h>
int main(void)
{
   void *buf;
   int handle, bytes;
   buf = malloc(10);
/*
Looks for a file in the current directory named TEST.$$$ and attempts to read 10 bytes from
it. To use this example you should create the file TEST.$$$.
 */
   if ((handle =
      open("TEST.$$$", O_RDONLY | O_BINARY, S_IWRITE | S_IREAD)) == -1)
   {
      printf("Error Opening File\n");
      exit(1);
   }
   if ((bytes = read(handle, buf, 10)) == -1) {
      printf("Read Failed.\n");
      exit(1);
   }
   else {
      printf("Read: %d bytes read.\n", bytes);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.15.31 setmode

Header File

io.h

Category

Input/output Routines

Prototype

```
int setmode(int handle, int amode);
```

Description

Sets mode of an open file.

setmode sets the mode of the open file associated with handle to either binary or text. The argument amode must have a value of either O_BINARY or O_TEXT, never both. (These symbolic constants are defined in fcntl.h.)

Return Value

setmode returns the previous translation mode if successful. On error it returns -1 and sets the global variable errno to

EINVAL	Invalid argument

Example

```
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
int main (int argc, char ** argv )
(
  FILE *fp;
  int newmode;
  long where;
  char buf[256];
  fp = fopen( argv[1], "r+" );
  if ( !fp )
  {
    printf( "Couldn't open %s\n", argv[1] );
    return -1;
  }
  newmode = setmode( fileno( fp ), O_BINARY );
  if ( newmode == -1 )
  {
    printf( "Couldn't set mode of %s\n", argv[1] );
    return -2
  }
  fp->flags |= _F_BIN;
  where = ftell( fp );
  printf ("file position: %d\n", where );
  fread( buf, 1, 1, fp );
  where = ftell ( fp );
  printf( "read %c, file position: %ld\n", *buf, where );
  fclose ( fp );
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.32 tell

Header File

io.h

Category

Input/output Routines

Prototype

```
long tell(int handle);
```

Description

Gets the current position of a file pointer.

tell gets the current position of the file pointer associated with handle and expresses it as the number of bytes from the beginning of the file.

Return Value

tell returns the current file pointer position. A return of -1 (**long**) indicates an error, and the global variable errno is set to

EBADF	Bad file number

Example

```
#include <string.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   char msg[] = "Hello world";
   if ((handle = open("TEST.$$$", O_CREAT | O_TEXT | O_APPEND)) == -1)
   {
      perror("Error:");
      return 1;
   }
   write(handle, msg, strlen(msg));
   printf("The file pointer is at byte %ld\n", tell(handle));
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.33 umask

Header File

io.h, sys\stat.h

Category

Input/output Routines

Prototype

```
unsigned umask(unsigned mode);
```

Description

Sets file read/write permission mask.

The umask function sets the access permission mask used by open and creat. Bits that are set in mode will be cleared in the access permission of files subsequently created by open and creat.

The mode can have one of the following values, defined in sys\stat.h:

S_IWRITE	Permission to write
S_IREAD	Permission to read
S_IREAD|S_IWRITE	Permission to read and write

Return Value

The previous value of the mask. There is no error return.

Example

```
#include <io.h>
#include <stdio.h>
#include <sys\stat.h>
#define FILENAME "TEST.$$$"
int main(void)
{
    unsigned oldmask;
    FILE *f;
    struct stat statbuf;
    /* Cause subsequent files to be created as read-only */
    oldmask = umask(S_IWRITE);
    printf("Old mask = 0x%x\n",oldmask);
    /* Create a zero-length file */
    if ((f = fopen(FILENAME,"w+")) == NULL)
    {
        perror("Unable to create output file");
        return (1);
    }
    fclose(f);
    /* Verify that the file is read-only */
    if (stat(FILENAME,&statbuf) != 0)
    {
        perror("Unable to get information about output file");
        return (1);
    }
    if (statbuf.st_mode & S_IWRITE)
        printf("Error! %s is writable!\n",FILENAME);
    else
        printf("Success! %s is not writable.\n",FILENAME);
    return (0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.15.34 unlock

Header File

io.h

Category

Input/output Routines

Prototype

```
int unlock(int handle, long offset, long length);
```

Description

Releases file-sharing locks.

unlock provides an interface to the operating system file-sharing mechanism. unlock removes a lock previously placed with a call to lock. To avoid error, all locks must be removed before a file is closed. A program must release all locks before completing.

Return Value

On success, unlock returns 0

O error, it returns -1.

Example

```
#include <io.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
int main(void)
{
   int handle, status;
   long length;
   handle = _sopen("c:\\autoexec.bat",O_RDONLY,SH_DENYNO,S_IREAD);
   if (handle < 0)
   {
       printf("_sopen failed\n");
       exit(1);
   }
   length = filelength(handle);
   status = lock(handle,0L,length/2);
   if (status == 0)
      printf("lock succeeded\n");
   else
      printf("lock failed\n");
   status = unlock(handle,0L,length/2);
   if (status == 0)
      printf("unlock succeeded\n");
   else
      printf("unlock failed\n");
   close(handle);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.15.35 vsscanf

Header File

io.h

Category

Memory and String Manipulation Routines

Prototype

```
int vsscanf(const char *buffer, const char *format, va_list arglist);
```

Description

Scans and formats input from a stream.

The v...scanf functions are known as alternate entry points for the ...scanf functions. They behave exactly like their ...scanf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

Note: For details on format specifiers, see Scanf Format Specifiers.

vsscanf scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specifier passed to vsscanf in the format string pointed to by format. Finally, vsscanf stores the formatted input at an address passed to it as an argument following format. There must be the same number of format specifiers and addresses as there are input fields.

vsscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See scanf for a discussion of possible causes.

Return Value

vsscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If vsscanf attempts to read at end-of-string, the return value is EOF.

Example

```
#include <stdio.h>
#include <stdarg.h>
char buffer[80] = "30 90.0 abc";
int vssf(char *fmt, ...)
{
   va_list  argptr;
   int cnt;
   fflush(stdin);
   va_start(argptr, fmt);
   cnt = vsscanf(buffer, fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
{
   int inumber;
   float fnumber;
   char string[80];
   vssf("%d %f %s", &inumber, &fnumber, string);
   printf("%d %f %s\n", inumber, fnumber, string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+	+	+

3.1.4.15.36 write

Header File

io.h

Category

Input/output Routines

Prototype

```
int write(int handle, void *buf, unsigned len);
```

Description

Writes to a file.

write writes a buffer of data to the file or device named by the given handle. handle is a file handle obtained from a creat, open, dup, or dup2 call.

This function attempts to write len bytes from the buffer pointed to by buf to the file associated with handle. Except when write is used to write to a text file, the number of bytes written to the file will be no more than the number requested. The maximum number of bytes that write can write is UINT_MAX -1, because UINT_MAX is the same as -1, which is the error return indicator for write. On text files, when write sees a linefeed (LF) character, it outputs a CR/LF pair. UINT_MAX is defined in limits.h.

If the number of bytes actually written is less than that requested, the condition should be considered an error and probably indicates a full disk. For disks or disk files, writing always proceeds from the current file pointer. For devices, bytes are sent directly to the device. For files opened with the O_APPEND option, the file pointer is positioned to EOF by write before writing the data.

Return Value

write returns the number of bytes written. A write to a text file does not count generated carriage returns. In case of error, write returns -1 and sets the global variable errno to one of the following values:

EACCES	Permission denied
EBADF	Bad file number

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <io.h>
#include <string.h>
int main(void)
{
   int handle;
   char string[40];
   int length, res;
/*
Create a file named "TEST.$$$" in the current directory and write a string to it.  If
"TEST.$$$" already exists, it will be overwritten.
 */
   if ((handle = open("TEST.$$$", O_WRONLY | O_CREAT | O_TRUNC,
                        S_IREAD | S_IWRITE)) == -1)
   {
      printf("Error opening file.\n");
      exit(1);
   }
   strcpy(string, "Hello, world!\n");
   length = strlen(string);
   if ((res = write(handle, string, length)) != length)
   {
      printf("Error writing to the file.\n");
      exit(1);
   }
   printf("Wrote %d bytes to the file.\n", res);
   close(handle);
```

```
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.16 limits.h

The following functions, macros, and classes are provided in `limits.h`:

Topics

Name	Description
CHAR_xxx #defines (see page 913)	**Header File** limits.h **Description**
INT_xxx #defines (see page 913)	**Header File** limits.h **Description** Maximum and minimum value for type int.
LONG_xxx #defines (see page 914)	**Header File** limits.h **Description** Maximum and minimum value for type long.
SCHAR_xxx #defines (see page 914)	**Header File** limits.h **Description**
SHRT_xxx #defines (see page 914)	**Header File** limits.h **Description**
Uxxxx_MAX #defines (see page 915)	**Header File** limits.h **Description**

3.1.4.16.1 CHAR_xxx #defines

Header File

limits.h

Description

Name	Meaning
CHAR_BIT	Type char, number of bits
CHAR_MAX	Type char, minimum value
CHAR_MIN	Type char, maximum value

These values are independent of whether type char is defined as signed or unsigned by default.

3.1.4.16.2 INT_xxx #defines

Header File

limits.h

Description

Maximum and minimum value for type int.

Name	Meaning
INT_MAX	Type int, maximum value
INT_MIN	Type int, minimum value

3.1.4.16.3 LONG_xxx #defines

Header File

limits.h

Description

Maximum and minimum value for type long.

Name	Meaning
LONG_MAX	Type long, maximum value
LONG_MIN	Type long, minimum value

3.1.4.16.4 SCHAR_xxx #defines

Header File

limits.h

Description

Name	Meaning
SCHAR_MAX	Type char, maximum value
SCHAR_MIN	Type char, minimum value

3.1.4.16.5 SHRT_xxx #defines

Header File

limits.h

Description

Name	Meaning
SHRT_MAX	Type short, maximum value
SHRT_MIN	Type short, minimum value

3.1.4.16.6 Uxxxx_MAX #defines

Header File

limits.h

Description

Name	Maximum value for type xxx
UCHAR_MAX	unsigned char
USHRT_MAX	unsigned short
UINT_MAX	unsigned integer
ULONG_MAX	unsigned long

3.1.4.17 locale.h

The following functions, macros, and classes are provided in `locale.h`:

Topics

Name	Description
localeconv (see page 915)	**Header File** locale.h **Category** Miscellaneous Routines **Prototype** `struct lconv *localeconv(void);` **Description** Queries the locale for numeric format. This function provides information about the monetary and other numeric formats for the current locale. The information is stored in a **struct** lconv type. The structure can only be modified by the setlocale. Subsequent calls to localeconv will update the lconv structure. The lconv structure is defined in locale.h. It contains the following fields:

setlocale, _wsetlocale (see page 917)	**Header File**
	locale.h
	Category
	Miscellaneous Routines
	Prototype
	`char *setlocale(int category, const char *locale);`
	`wchar_t * _wsetlocale( int category, const wchar_t *locale);`
	Description
	Use the setlocale to select or query a locale.
	C++ Builder supports all locales supported in Win95/98/2000 operating systems. See your system documentation for details.
	The possible values for the category argument are as follows:

3.1.4.17.1 localeconv

Header File

locale.h

Category

Miscellaneous Routines

Prototype

```
struct lconv *localeconv(void);
```

Description

Queries the locale for numeric format.

This function provides information about the monetary and other numeric formats for the current locale. The information is stored in a **struct** lconv type. The structure can only be modified by the setlocale. Subsequent calls to localeconv will update the lconv structure.

The lconv structure is defined in locale.h. It contains the following fields:

char	**Decimal point used in nonmonetary formats. This can never be an empty string.**
char	Separator used to group digits to the left of the decimal point. Not used with monetary quantities.
char	Size of each group of digits. Not used with monetary quantities. See the value listing table below.
char	International monetary symbol in the current locale. The symbol format is specified in the ISO 4217 Codes for the Representation of Currency and Funds.
char	Local monetary symbol for the current locale.
char	Decimal point used to format monetary quantities.
char	Separator used to group digits to the left of the decimal point for monetary quantities.
char	Size of each group of digits used in monetary quantities. See the value listing table below.
char	String indicating nonnegative monetary quantities.
char	String indicating negative monetary quantities.
char	Number of digits after the decimal point that are to be displayed in an internationally formatted monetary quantity.
char	Number of digits after the decimal point that are to be displayed in a formatted monetary quantity.
char	Set to 1 if currency_symbol precedes a nonnegative formatted monetary quantity. If currency_symbol is after the quantity, it is set to 0.
char	Set to 1 if currency_symbol is to be separated from the nonnegative formatted monetary quantity by a space. Set to 0 if there is no space separation.
char	Set to 1 if currency_symbol precedes a negative formatted monetary quantity. If currency_symbol is after the quantity, set to 0.
char	Set to 1 if currency_symbol is to be separated from the negative formatted monetary quantity by a space. Set to 0 if there is no space separation.
char	Indicate where to position the positive sign in a nonnegative formatted monetary quantity.
char	Indicate where to position the positive sign in a negative formatted monetary quantity.

Any of the above strings (except decimal_point) that is empty " " is not supported in the current locale. The nonstring **char** elements are nonnegative numbers. Any nonstring **char** element that is set to CHAR_MAX indicates that the element is not supported in the current locale.

The grouping and mon_grouping elements are set and interpreted as follows:

CHAR_MAX	No further grouping is to be performed.
0	The previous element is to be used repeatedly for the remainder of the digits.
any other integer	Indicates how many digits make up the current group. The next element is read to determine the size of the next group of digits before the current group.

The p_sign_posn and n_sign_posn elements are set and interpreted as follows:

0	Use parentheses to surround the quantity and currency_symbol.
1	Sign string precedes the quantity and currency_symbol.
2	Sign string succeeds the quantity and currency_symbol.
3	Sign string immediately precedes the quantity and currency_symbol.
4	Sign string immediately succeeds the quantity and currency_symbol.

Return Value

Returns a pointer to the the filled-in structure of type **struct** lconv. The values in the structure will change whenever setlocale modifies the LC_MONETARY or LC_NUMERIC categories.

Example

```
#include <locale.h>
#include <stdio.h>
int main(void)
{
   struct lconv ll;
   struct lconv *conv = &ll;
/* read the locality conversion structure */
   conv = localeconv();
/* display the structure */
  printf("Decimal Point                : %s\n", conv-> decimal_point);
  printf("Thousands Separator          : %s\n", conv-> thousands_sep);
  printf("Grouping                     : %s\n", conv-> grouping);
  printf("International Currency symbol : %s\n", conv-> int_curr_symbol);
  printf("$ thousands separator        : %s\n", conv-> mon_thousands_sep);
  printf("$ grouping                   : %s\n", conv-> mon_grouping);
  printf("Positive sign                : %s\n", conv-> positive_sign);
  printf("Negative sign                : %s\n", conv-> negative_sign);
  printf("International fraction digits : %d\n", conv-> int_frac_digits);
  printf("Fraction digits              : %d\n", conv-> frac_digits);
  printf("Positive $ symbol precedes   : %d\n", conv-> p_cs_precedes);
  printf("Positive sign space separation: %d\n", conv-> p_sep_by_space);
  printf("Negative $ symbol precedes   : %d\n", conv-> n_cs_precedes);
  printf("Negative sign space separation: %d\n", conv-> n_sep_by_space);
  printf("Positive sign position       : %d\n", conv-> p_sign_posn);
  printf("Negative sign position       : %d\n", conv-> n_sign_posn);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.17.2 setlocale, _wsetlocale

Header File

locale.h

Category

Miscellaneous Routines

Prototype

```
char *setlocale(int category, const char *locale);
wchar_t * _wsetlocale( int category, const wchar_t *locale);
```

Description

Use the setlocale to select or query a locale.

C++ Builder supports all locales supported in Win95/98/2000 operating systems. See your system documentation for details.

The possible values for the category argument are as follows:

LC_ALL	Affects all the following categories
LC_COLLATE	Affects strcoll and strxfrm
LC_CTYPE	Affects single-byte character handling functions. The mbstowcs and mbtowc functions are not affected.
LC_MONETARY	Affects monetary formatting by the localeconv function
LC_NUMERIC	Affects the decimal point of non-monetary data formatting. This includes the printf family of functions, and the information returned by localeconv.
LC_TIME	Affects strftime

The locale argument is a pointer to the name of the locale or named locale category. Passing a NULL pointer returns the current locale in effect. Passing a pointer that points to a null string requests setlocale to look for environment variables to determine which locale to set. The locale names are **not** case sensitive.

When setlocale is unable to honor a locale request, the preexisting locale in effect is unchanged and a null pointer is returned.

If the locale argument is a NULL pointer, the locale string for the category is returned. If category is LC_ALL, a complete locale string is returned. The structure of the complete locale string consists of the names of all the categories in the current locale concatenated and separated by semicolons. This string can be used as the locale parameter when calling setlocale with any of the LC_xxx values. This will reinstate all the locale categories that are named in the complete locale string, and allows saving and restoring of locale states. If the complete locale string is used with a single category, for example, LC_TIME, only that category will be restored from the locale string.

If an empty string "" is used as the locale parameter an implementation-defined locale is used. This is the ANSI C specified behavior.

To take advantage of dynamically loadable locales in your application, define _ _USELOCALES_ _ for each module. If _ _USELOCALES_ _ is not defined, all locale-sensitive functions and macros will work only with the default C locale.

If a NULL pointer is used as the argument for the locale parameter, setlocale returns a string that specifies the current locale in effect. If the category parameter specifies a single category, such as LC_COLLATE, the string pointed to will be the name of that category. If LC_ALL is used as the category parameter then the string pointed to will be a full locale string that will indicate the name of each category in effect.

```
.
.
.
localenameptr = setlocale( LC_COLLATE, NULL );
if (localenameptr)
printf( "%s\n", localenameptr );
.
.
```

.

The output here will be one of the module names together with the specified code page. For example, the output could be LC_COLLATE = English_United States.437.

.

.

.

```
localenameptr = setlocale( LC_ALL, NULL );
if (localenameptr)
printf( "%s\n", localenameptr );
```

.

.

.

An example of the output here could be the following:

```
LC_COLLATE=English_United States.437;
LC_TIME=English_United States.437;
LC_CTYPE=English_United States.437;
```

Each category in this full string is delimited by a semicolon. This string can be copied and saved by an application and then used again to restore the same locale categories at another time. Each delimited name corresponds to the locale category constants defined in locale.h. Therefore, the first name is the name of the LC_COLLATE category, the second is the LC_CTYPE category, and so on. Any other categories named in the locale.h header file are reserved for future implementation.

To set all default categories for the specified French locale:

```
setlocale( LC_ALL, "French_France.850" );
```

To find out which code page is currently being used:

```
localenameptr = setlocale( LC_ALL, NULL );
```

Return value

If selection is successful, setlocale returns a pointer to a string that is associated with the selected category (or possibly all categories) for the new locale.

If UNICODE is defined, _wsetlocale returns a **wchar_t** string.

On failure, a NULL pointer is returned and the locale is unchanged. All other possible returns are discussed in the Remarks section above.

Example

```
#include <locale.h>
#include <stdio.h>
int main(void)
{
   char *old_locale;
   /* The only locale supported in CodeGear C++ is "C" */
   old_locale = setlocale(LC_ALL,"C");
   printf("Old locale was %s\n",old_locale);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
setlocale	+	+	+	+
_wsetlocale		+		

3.1.4.18 malloc.h

The following functions, macros, and classes are provided in `malloc.h`:

Topics

Name	Description
alloca (see page 920)	**Header File** malloc.h **Category** Memory Routines **Prototype** `void *alloca(size_t size);` **Description** Allocates temporary stack space. alloca allocates size bytes on the stack; the allocated space is automatically freed up when the calling function exits. The use of alloca is not encouraged. In the **try**-block of a C++ program the alloca function should never be used. If an exception is thrown, any values placed on the stack by alloca will be corrupted. **Return Value** If enough stack space is available, alloca returns a pointer to the allocated stack area. Otherwise, it returns NULL. **Example**

3.1.4.18.1 alloca

Header File

malloc.h

Category

Memory Routines

Prototype

```
void *alloca(size_t size);
```

Description

Allocates temporary stack space.

alloca allocates size bytes on the stack; the allocated space is automatically freed up when the calling function exits.

The use of alloca is not encouraged. In the **try**-block of a C++ program the alloca function should never be used. If an exception is thrown, any values placed on the stack by alloca will be corrupted.

Return Value

If enough stack space is available, alloca returns a pointer to the allocated stack area. Otherwise, it returns NULL.

Example

```
#include <malloc.h>
#include <stdio.h>
```

```
#include <stdlib.h>
void test(int a)
{
   char *newstack;
   int len = a;
   char dummy[1];
dummy[0] = 0;          /* force good stack frame */
   printf("SP before calling alloca(0x%X) = 0x%X\n",len,_SP);
   newstack = (char *) alloca(len);
printf("SP after calling alloca = 0x%X\n",_SP);
   if (newstack)
      printf("Alloca(0x%X) returned %p\n",len,newstack);
   else
      printf("Alloca(0x%X) failed\n",len);
}
void main()
{
   test(256);
   test(16384);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.19 math.h

The following functions, macros, and classes are provided in `math.h`:

Topics

Name	Description
HUGE_VAL #defines (see page 931)	**Header File** math.h **Description** Overflow value for math functions.
M_E, M_LOGxxx, M_LNxx #defines (see page 932)	**Header File** math.h **Description** The constant values for logarithm functions.
M_SQRTxx #defines (see page 932)	**Header File** math.h **Description** Constant values for square roots of 2.
PI constants (see page 932)	**Header File** math.h **Description** Common constants of pi

atof, _wtof (see page 933)	**Header File**
	stdlib.h, math.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`double atof(const char *s);`
	`double _wtof(const wchar_t *s);`
	Description
	Converts a string to a floating-point number.
	• atof converts a string pointed to by s to **double**; this function recognizes the character representation of a floating-point number, made up of the following:
	• An optional string of tabs and spaces
	• An optional sign
	• A string of digits and an optional decimal point (the digits can be on both sides of the decimal point)
	• An optional e or E followed by an optional signed integer
	The characters must match this generic... more (see page 933)

_atold, _wtold (see page 933)	**Header File**
	math.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`long double _atold(const char *s);`
	`long double _wtold(const wchar_t *s);`
	Description
	Converts a string to a long double.
	_wtold is the wide-character version. It converts a wide-character string to a long double.
	_atof is the floating-point version of _atold.
	_atold converts a string pointed to by s to a **long double**; this functions recognizes:
	An optional string of tabs and spaces
	An optional sign
	A string of digits and an optional decimal point
	An optional e or E followed by an optional signed integer
	The first unrecognized character ends... more (see page 933)

_i64toa, _i64tow (see page 934)	**Header File**
	math.h, stdlib.h
	Category
	Conversion Routines, Math Routines,
	Prototype
	`char *_i64toa(__int64 value, char *string, int radix);`
	`wchar_t *_i64tow(__int64 value, wchar_t *string, int radix);`
	Description
	_i64toa converts an **__int64** to a string. _i64tow is the unicode version. It converts a **__int64** to a wide-character string.
	These functions convert value to a null-terminated string and store the result in string. value is an **__int64**.
	radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus sign... more (see page 934)

_matherr, _matherrl (see page 935)	**Header File**
	math.h
	Category
	Diagnostic Routines, Math Routines
	Prototype
	`int _matherr(struct _exception *e);`
	`int _matherrl(struct _exceptionl *e);`
	Description
	User-modifiable math error handler.
	_matherr is called when an error is generated by the math library.
	_matherrl is the **long double** version; it is called when an error is generated by the long double math functions.
	_matherr and _matherrl each serve as a user hook (a function that can be customized by the user) that you can replace by writing your own math error-handling routine.
	_matherr and _matherrl are useful for information on trapping domain and range errors caused by the... more (see page 935)

abs (see page 936)	**Header File** stdlib.h, math.h **Category** Math Routines, Inline Routines **Prototype** `int abs(int x);` **Description** Returns the absolute value of an integer. abs returns the absolute value of the integer argument x. If abs is called when stdlib.h has been included, it's treated as a macro that expands to inline code. If you want to use the abs function instead of the macro, include #undef abs in your program, after the #include <stdlib.h>. **Return Value** The abs function returns an integer in the range of 0 to INT_MAX, with the exception that an argument with the value INT_MIN is returned... more (see page 936)

acos, acosl (see page 937)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double acos(double x);`
	`long double acosl(long double x);`
	Description
	Calculates the arc cosine.
	acos returns the arc cosine of the input value.
	acosl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Arguments to acos and acosl must be in the range -1 to 1, or else acos and acosl return NAN and set the global variable errno to:
	EDOM Domain error
	Return Value
	acos and acosl of an argument between -1 and +1 return a value in the range 0 to pi. Error... more (see page 937)

asin, asinl (see page 938)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double asin(double x);`
	`long double asinl(long double x);`
	Description
	Calculates the arc sine.
	asin of a real argument returns the arc sine of the input value.
	asinl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Real arguments to asin and asinl must be in the range -1 to 1, or else asin and asinl return NAN and set the global variable errno to
	EDOM Domain error
	Return Value
	asin and asinl of a real argument return a value in the range -pi/2 to... more (see page 938)

atan, atanl (see page 939)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double atan(double x);`
	`long double atanl(long double x);`
	Description
	Calculates the arc tangent.
	atan calculates the arc tangent of the input value.
	atanl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	atan and atanl of a real argument return a value in the range -pi/2 to pi/2. Error handling for these functions can be modified through the functions _matherr and _matherrl.
	Example

atan2, atan2l (see page 940)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double atan2(double y, double x);`
	`long double atan2l(long double y, long double x);`
	Description
	Calculates the arc tangent of y/x.
	atan2 returns the arc tangent of y/x; it produces correct results even when the resulting angle is near pi/2 or -pi/2 (x near 0). If both x and y are set to 0, the function sets the global variable errno to EDOM, indicating a domain error.
	atan2l is the **long double** version; it takes **long double** arguments and returns a **long double** result.
	Return Value
	atan2 and atan2l return a value in... more (see page 940)

cabs, cabsl (see page 940)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double cabs(struct complex z);`
	`long double cabsl(struct _complexl z);`
	Description
	cabs calculates the absolute value of a complex number. cabs is a macro that calculates the absolute value of z, a complex number. z is a structure with type complex; the structure is defined in math.h as
	`struct complex {`
	`double x, y;`
	`};`
	where x is the real part, and y is the imaginary part.
	Calling cabs is equivalent to calling sqrt with the real and imaginary components of z, as shown here:
	`sqrt(z.x * z.x + z.y * z.y)`
	cabsl... more (see page 940)

ceil, ceill (see page 942)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double ceil(double x);`
	`long double ceill(long double x);`
	Description
	Rounds up.
	ceil finds the smallest integer not less than x.
	ceill is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	These functions return the integer found as a **double** (ceil) or a **long double** (ceill).
	Example

cos, cosl (see page 943)	**Header File**
	math.h
	Category
	Math Routines, Inline Routines
	Prototype
	`double cos(double x);`
	`long double cosl(long double x);`
	Description
	Calculates the cosine of a value.
	cos computes the cosine of the input value. The angle is specified in radians.
	cosl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	cos of a real argument returns a value in the range -1 to 1. Error handling for these functions can be modified through _matherr (or _matherrl).
	Example

cosh, coshl (see page 943)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double cosh(double x);`
	`long double coshl(long double x);`
	Description
	Calculates the hyperbolic cosine of a value.
	cosh computes the hyperbolic cosine:
	coshl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	cosh returns the hyperbolic cosine of the argument.
	When the correct value would create an overflow, these functions return the value HUGE_VAL (cosh) or _LHUGE_VAL (coshl) with the appropriate sign, and the global variable errno is set to ERANGE. Error handling for these functions can be modified through the functions _matherr... more (see page 943)

exp, expl (see page 944)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double exp(double x);`
	`long double expl(long double x);`
	Description
	Calculates the exponential e to the x.
	expl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	exp returns e to the x.
	Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value overflows, exp returns the value HUGE_VAL and expl returns _LHUGE_VAL. Results of excessively large magnitude cause the global variable errno to be set to

fabs, fabsl (see page 945)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double fabs(double x);`
	`long double fabsl(long double x);`
	Description
	Returns the absolute value of a floating-point number.
	fabs calculates the absolute value of x, a double. fabsl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	fabs and fabsl return the absolute value of x.
	Example

floor, floorl (see page 946)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double floor(double x);`
	`long double floorl(long double x);`
	Description
	Rounds down.
	floor finds the largest integer not greater than x.
	floorl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	floor returns the integer found as a **double**. floorl returns the integer found as a **long double**.
	Portability

fmod, fmodl (see page 946)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double fmod(double x, double y);`
	`long double fmodl(long double x, long double y);`
	Description
	Calculates x modulo y, the remainder of x/y.
	fmod calculates x modulo y (the remainder f, where x = ay + f for some integer a, and 0 < f < y).
	fmodl is the **long double** version; it takes **long double** arguments and returns a **long double** result.
	Return Value
	fmod and fmodl return the remainder f where x = ay + f (as described above). When y = 0, fmod and fmodl return 0.
	Example

frexp, frexpl (see page 947)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double frexp(double x, int *exponent);`
	`long double frexpl(long double x, int *exponent);`
	Description
	Splits a number into mantissa and exponent.
	frexp calculates the mantissa m (a **double** greater than or equal to 0.5 and less than 1) and the integer value n such that x (the original **double** value) equals m * 2n. frexp stores n in the integer that exponent points to.
	frexpl is the **long double** version; it takes a **long double** argument for x and returns a **long double** result.
	Return Value
	frexp and frexpl return the mantissa m.... more (see page 947)

hypot, hypotl (see page 948)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double hypot(double x, double y);`
	`long double hypotl(long double x, long double y);`
	Description
	Calculates hypotenuse of a right triangle.
	hypot calculates the value z where
	z2 = x2 + y2 and z >= 0
	This is equivalent to the length of the hypotenuse of a right triangle, if the lengths of the two sides are x and y.
	hypotl is the **long double** version; it takes **long double** arguments and returns a **long double** result.
	Return Value
	On success, these functions return z, a **double** (hypot) or a **long double**)... more (see page 948)

ldexp, ldexpl (see page 949)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double ldexp(double x, int exp);`
	`long double ldexpl(long double x, int exp);`
	Description
	Calculates x * 2^exp.
	lexpl is the **long double** version; it takes a **long double** argument for x and returns a **long double** result.
	Return Value
	On success, ldexp (or ldexpl) returns the value it calculated, x * 2^exp. Error handling for these routines can be modified through the functions _matherr and _matherrl.
	Example

ldiv (see page 950)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`ldiv_t ldiv(long int numer, long int denom);`
	Description
	Divides two **longs**, returning quotient and remainder.
	ldiv divides two **longs** and returns both the quotient and the remainder as an ldiv_t type. numer and denom are the numerator and denominator, respectively.
	The ldiv_t type is a structure of **long**s defined in stdlib.h as follows:
	`typedef struct {`
	`long int quot; /* quotient */`
	`long int rem; /* remainder */`
	`} ldiv_t;`
	Return Value
	ldiv returns a structure whose elements are quot (the quotient) and rem (the remainder).
	Example

log, logl (see page 950)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double log(double x);`
	`long double logl(long double x);`
	Description
	Calculates the natural logarithm of x.
	log calculates the natural logarithm of x.
	logl is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	On success, log and logl return the value calculated, ln(x).
	errno

log10, log10l (see page 951)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double log10(double x);`
	`long double log10l(long double x);`
	Description
	log10 calculates the base ten logarithm of x.
	log10l is the **long double** version; it takes a **long double** argument and returns a **long double** result.
	Return Value
	On success, log10 (or log10l) returns the calculated value log base ten of x.
	If the argument x passed to these functions is real and less than 0, the global variable errno is set to

modf, modfl (see page 952)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double modf(double x, double *ipart);`
	`long double modfl(long double x, long double *ipart);`
	Description
	Splits a **double** or **long double** into integer and fractional parts.
	modf breaks the **double** x into two parts: the integer and the fraction. modf stores the integer in ipart and returns the fraction.
	modfl is the **long double** version; it takes **long double** arguments and returns a **long double** result.
	Return Value
	modf and modfl return the fractional part of x.
	Example

poly, polyl (see page 953)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double poly(double x, int degree, double coeffs[]);`
	`long double polyl(long double x, int degree, long double coeffs[]);`
	Description
	Generates a polynomial from arguments.
	poly generates a polynomial in x, of degree degree, with coefficients coeffs[0], coeffs[1], ..., coeffs[degree]. For example, if n = 4, the generated polynomial is:
	polyl is the **long double** version; it takes **long double** arguments and returns a **long double** result.
	Return Value
	poly and polyl return the value of the polynomial as evaluated for the given x.
	Example

pow, powl (see page 954)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double pow(double x, double y);`
	`long double powl(long double x, long double y);`
	Description
	Calculates x to the power of y.
	powl is the **long double** version; it takes **long double** arguments and returns a **long double** result.
	Return Value
	On success, pow and powl return the value calculated of x to the power of y.
	Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value would overflow, the functions return the value HUGE_VAL (pow) or _LHUGE_VAL (powl). Results of excessively large magnitude can... more (see page 954)

pow10, pow10l (see page 955)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double pow10(int p);`
	`long double pow10l(int p);`
	Description
	Calculates 10 to the power of p.
	pow10l is the **long double** version; it takes **long double** arguments and returns a **long double** result.
	Return Value
	On success, pow10 returns the value calculated, 10 to the power of p and pow10l returns a **long double** result.
	The result is actually calculated to **long double** accuracy. All arguments are valid, although some can cause an underflow or overflow.
	Example

sin, sinl (see page 955)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double sin(double x);`
	`long double sinl(long double x);`
	Description
	Calculates sine.
	sin computes the sine of the input value. Angles are specified in radians.
	sinl is the **long double** version; it takes a l**ong double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.
	Return Value
	sin and sinl return the sine of the input value.
	Example

sinh, sinhl (see page 956)	**Header File**
	math.h
	Category
	Math Routines, Inline Routines
	Prototype
	`double sinh(double x);`
	`long double sinhl(long double x);`
	Description
	Calculates hyperbolic sine.
	sinh computes the hyperbolic sine.
	sinl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for sinh and sinhl can be modified through the functions _matherr and _matherrl.
	Return Value
	sinh and sinhl return the hyperbolic sine of x.
	When the correct value overflows, these functions return the value HUGE_VAL (sinh) or _LHUGE_VAL (sinhl) of appropriate sign. Also, the global variable errno is set to ERANGE.
	Example

sqrt, sqrtl (see page 957)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double sqrt(double x);`
	`long double sqrtl(long double x);`
	Description
	Calculates the positive square root.
	sqrt calculates the positive square root of the argument x.
	sqrtl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.
	Return Value
	On success, sqrt and sqrtl return the value calculated, the square root of x. If x is real and positive, the result is positive. If x is real and negative, the global variable errno... more (see page 957)

tan, tanl (see page 958)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double tan(double x);`
	`long double tanl(long double x);`
	Description
	Calculates the tangent.
	tan calculates the tangent. Angles are specified in radians.
	tanl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these routines can be modified through the functions _matherr and _matherrl..
	Return Value
	tan and tanl return the tangent of x, sin(x)/cos(x).
	Example

tanh, tanhl (see page 958)	**Header File**
	math.h
	Category
	Math Routines
	Prototype
	`double tanh(double x);`
	`long double tanhl(long double x);`
	Description
	Calculates the hyperbolic tangent.
	tanh computes the hyperbolic tangent, sinh(x)/cosh(x).
	tanhl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.
	Return Value
	tanh and tanhl return the hyperbolic tangent of x.
	Example

3.1.4.19.1 HUGE_VAL #defines

Header File

math.h

Description

Overflow value for math functions.

3.1.4.19.2 M_E, M_LOGxxx, M_LNxx #defines

Header File

math.h

Description

The constant values for logarithm functions.

Name	Meaning
M_E	The value of e
M_LOG2E	The value of log(e)
M_LOG10E	The value of log10(e)
M_LN2	The value of ln(2)
M_LN10	The value of ln(10)

3.1.4.19.3 M_SQRTxx #defines

Header File

math.h

Description

Constant values for square roots of 2.

Name	Meaning
M_SQRT2	Square root of 2
M_SQRT_2	1/2 the square root of 2

3.1.4.19.4 PI constants

Header File

math.h

Description

Common constants of pi

Name	Meaning
M_PI	pi
M_PI_2	One-half pi
M_PI_4	One-fourth pi
M_1_PI	One divided by pi
M_2_PI	Two divided by pi
M_1_SQRTPI	One divided by the square root of pi
M_2_SQRTPI	Two divided by the square root of pi

3.1.4.19.5 atof, _wtof

Header File

stdlib.h, math.h

Category

Conversion Routines, Math Routines

Prototype

```
double atof(const char *s);
double _wtof(const wchar_t *s);
```

Description

Converts a string to a floating-point number.

- atof converts a string pointed to by s to **double**; this function recognizes the character representation of a floating-point number, made up of the following:
- An optional string of tabs and spaces
- An optional sign
- A string of digits and an optional decimal point (the digits can be on both sides of the decimal point)
- An optional e or E followed by an optional signed integer

The characters must match this generic format:

[whitespace] [sign] [ddd] [.] [ddd] [e|E[sign]ddd]

atof also recognizes +INF and -INF for plus and minus infinity, and +NAN and -NAN for not-a-number.

In this function, the first unrecognized character ends the conversion.

The functions strtod and _strtold are similar to atof and provide better error detection, and hence are preferred in some applications.

Return Value

Returns the converted value of the input string.

If there is an overflow, atof returns plus or minus HUGE_VAL (or _LHUGE_VAL), errno is set to ERANGE (Result out of range), and _matherr (or _matherrl) is not called.

3.1.4.19.6 _atold, _wtold

Header File

math.h

Category

Conversion Routines, Math Routines

Prototype

```
long double _atold(const char *s);
long double _wtold(const wchar_t *s);
```

Description

Converts a string to a long double.

_wtold is the wide-character version. It converts a wide-character string to a long double.

_atof is the floating-point version of _atold.

_atold converts a string pointed to by s to a **long double**; this functions recognizes:

An optional string of tabs and spaces

An optional sign

A string of digits and an optional decimal point

An optional e or E followed by an optional signed integer

The first unrecognized character ends the conversion. There are no provisions for overflow.

The functions strtod and _strtold are similar to atof and _atold; they provide better error detection, and hence are preferred in some applications.

Return Value

Returns the converted value of the input string.

If there is an overflow, _atold returns plus or minus HUGE_VAL (or _LHUGE_VAL), errno is set to ERANGE (Result out of range), and _matherr (or _matherrl) is not called.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_atold		+		
_wtold		+		

3.1.4.19.7 _i64toa, _i64tow

Header File

math.h, stdlib.h

Category

Conversion Routines, Math Routines,

Prototype

```
char *_i64toa(__int64 value, char *string, int radix);
wchar_t *_i64tow(__int64 value, wchar_t *string, int radix);
```

Description

_i64toa converts an **__int64** to a string. _i64tow is the unicode version. It converts a **__int64** to a wide-character string.

These functions convert value to a null-terminated string and store the result in string. value is an **__int64**.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus sign (**-**).

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character (\0). These functions can return up to 33 bytes.

Return Value

Returns a pointer to string.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_i64toa		+		
_i64tow		+		

3.1.4.19.8 _matherr, _matherrl

Header File

math.h

Category

Diagnostic Routines, Math Routines

Prototype

```
int _matherr(struct _exception *e);
int _matherrl(struct _exceptionl *e);
```

Description

User-modifiable math error handler.

_matherr is called when an error is generated by the math library.

_matherrl is the **long double** version; it is called when an error is generated by the long double math functions.

_matherr and _matherrl each serve as a user hook (a function that can be customized by the user) that you can replace by writing your own math error-handling routine.

_matherr and _matherrl are useful for information on trapping domain and range errors caused by the math functions. They do not trap floating-point exceptions, such as division by zero. See signal for information on trapping such errors.

You can define your own _matherr or _matherrl routine to be a custom error handler (such as one that catches and resolves certain types of errors); this customized function overrides the default version in the C library. The customized _matherr or _matherrl should return 0 if it fails to resolve the error, or nonzero if the error is resolved. When _matherr or _matherrl return nonzero, no error message is printed and the global variable errno is not changed.

Here are the _exception and _exceptionl structures (defined in math.h):

```
struct _exception {
int type;
char *name;
double arg1, arg2, retval;
};
struct _exceptionl {
int type;
char *name;
long double arg1, arg2, retval;
};
```

The members of the _exception and _exceptionl structures are shown in the following table:

type	The type of mathematical error that occurred; an enum type defined in the typedef _mexcep (see definition after this list).
name	A pointer to a null-terminated string holding the name of the math library function that resulted in an error.
arg1, arg2	The arguments (passed to the function that name points to) caused the error; if only one argument was passed to the function, it is stored in arg1.
retval	The default return value for _matherr (or _matherrl); you can modify this value.

The **typedef** _mexcep, also defined in math.h, enumerates the following symbolic constants representing possible mathematical errors:

DOMAIN	Argument was not in domain of function, such as log(-1).
SING	Argument would result in a singularity, such as pow(0, -2).
OVERFLOW	Argument would produce a function result greater than DBL_MAX (or LDBL_MAX), such as exp(1000).
UNDERFLOW	Argument would produce a function result less than DBL_MIN (or LDBL_MIN), such as exp(-1000).
TLOSS	Argument would produce function result with total loss of significant digits, such as sin(10e70).

The macros DBL_MAX, DBL_MIN, LDBL_MAX, and LDBL_MIN are defined in float.h

The source code to the default _matherr and _matherrl is on the C++Builder distribution disks.

The UNIX-style _matherr and _matherrl default behavior (printing a message and terminating) is not ANSI compatible. If you want a UNIX-style version of these routines, use MATHERR.C and MATHERRL.C provided on the C++Builder distribution disks.

Example

```
#include <math.h>
#include <string.h>
#include <stdio.h>
int _matherr (struct _exception *a)
{
  if (a->type == DOMAIN)
    if (!strcmp(a->name,"sqrt")) {
      a->retval = sqrt (-(a->arg1));
    return 1;
    }
  return 0;
}
int main(void)
{
  double x = -2.0, y;
  y = sqrt(x);
  printf("_Matherr corrected value: %lf\n",y);
  return 0;
}
```

3.1.4.19.9 abs

Header File

stdlib.h, math.h

Category

Math Routines, Inline Routines

Prototype

```
int abs(int x);
```

Description

Returns the absolute value of an integer.

abs returns the absolute value of the integer argument x. If abs is called when stdlib.h has been included, it's treated as a macro that expands to inline code.

If you want to use the abs function instead of the macro, include

#undef abs

in your program, after the #include <stdlib.h>.

Return Value

The abs function returns an integer in the range of 0 to INT_MAX, with the exception that an argument with the value INT_MIN is returned as INT_MIN. The values for INT_MAX and INT_MIN are defined in header file limit.h.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
  int number = -1234;
  printf("number: %d  absolute value: %d\n", number, abs(number));
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.19.10 acos, acosl

Header File

math.h

Category

Math Routines

Prototype

```
double acos(double x);
```

```
long double acosl(long double x);
```

Description

Calculates the arc cosine.

acos returns the arc cosine of the input value.

acosl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Arguments to acos and acosl must be in the range -1 to 1, or else acos and acosl return NAN and set the global variable errno to:

EDOM Domain error

Return Value

acos and acosl of an argument between -1 and +1 return a value in the range 0 to pi. Error handling for these routines can be modified through the functions _matherr_matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
  double result;
  double x = 0.5;
  result = acos(x);
  printf("The arc cosine of %lf is %lf\n", x, result);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
acos	+	+	+	+
acosl		+	+	+

3.1.4.19.11 asin, asinl

Header File

math.h

Category

Math Routines

Prototype

```
double asin(double x);
```

```
long double asinl(long double x);
```

Description

Calculates the arc sine.

asin of a real argument returns the arc sine of the input value.

asinl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Real arguments to asin and asinl must be in the range -1 to 1, or else asin and asinl return NAN and set the global variable errno to

EDOM Domain error

Return Value

asin and asinl of a real argument return a value in the range -pi/2 to pi/2. Error handling for these functions may be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
```

```
   double x = 0.5;
   result = asin(x);
   printf("The arc sin of %lf is %lf\n", x, result);
  return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
asin	+	+	+	+
asinl		+	+	+

3.1.4.19.12 atan, atanl

Header File

math.h

Category

Math Routines

Prototype

```
double atan(double x);
```

```
long double atanl(long double x);
```

Description

Calculates the arc tangent.

atan calculates the arc tangent of the input value.

atanl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

atan and atanl of a real argument return a value in the range -pi/2 to pi/2. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 0.5;
   result = atan(x);
   printf("The arc tangent of %lf is %lf\n", x, result);
return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
atan	+	+	+	+
atanl		+	+	+

3.1.4.19.13 atan2, atan2l

Header File

math.h

Category

Math Routines

Prototype

```
double atan2(double y, double x);
```

```
long double atan2l(long double y, long double x);
```

Description

Calculates the arc tangent of y/x.

atan2 returns the arc tangent of y/x; it produces correct results even when the resulting angle is near pi/2 or -pi/2 (x near 0). If both x and y are set to 0, the function sets the global variable errno to EDOM, indicating a domain error.

atan2l is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

atan2 and atan2l return a value in the range -pi to pi. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 90.0, y = 45.0;
   result = atan2(y, x);
   printf("The arc tangent ratio of %lf is %lf\n", (y / x), result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
atan2	+	+	+	+
atan2l		+	+	+

3.1.4.19.14 cabs, cabsl

Header File

math.h

Category

Math Routines

Prototype

```
double cabs(struct complex z);
```

```
long double cabsl(struct _complexl z);
```

Description

cabs calculates the absolute value of a complex number. cabs is a macro that calculates the absolute value of z, a complex number. z is a structure with type complex; the structure is defined in math.h as

```
struct complex {
double x, y;
};
```

where x is the real part, and y is the imaginary part.

Calling cabs is equivalent to calling sqrt with the real and imaginary components of z, as shown here:

```
sqrt(z.x * z.x + z.y * z.y)
```

cabsl is the **long double** version; it takes a structure with type _complexl as an argument, and returns a **long double** result. The structure is defined in math.h as

```
struct _complexl {
long double x, y;
};
```

Return Value

cabs (or cabsl) returns the absolute value of z, a double. On overflow, cabs (or cabsl) returns HUGE_VAL (or _LHUGE_VAL) and sets the global variable errno to

ERANGE	Result out of range

Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
#ifdef __cplusplus
  #include <complex.h>
#endif
#ifdef __cplusplus /* if C++, use class complex */
  void print_abs(void)
  {
    complex<float> z(1.0, 2.0);
    double  absval;
    absval = abs(z);
    printf("The absolute value of %.2lfi %.2lfj is %.2lf",
      real(z), imag(z), absval);
  }
#else  /* below function is for C (and not C++) */
  void print_abs(void)
  {
    struct complex z;
    double absval;
    z.x = 2.0;
    z.y = 1.0;
    absval = cabs(z);
    printf("The absolute value of %.2lfi %.2lfj is %.2lf",
      z.x, z.y, absval);
  }
#endif
int main(void)
{
```

```
  print_abs();
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
cabs		+		
cabsl		+		

3.1.4.19.15 ceil, ceill

Header File

math.h

Category

Math Routines

Prototype

```
double ceil(double x);
```

```
long double ceill(long double x);
```

Description

Rounds up.

ceil finds the smallest integer not less than x.

ceill is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

These functions return the integer found as a **double** (ceil) or a **long double** (ceill).

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double number = 123.54;
   double down, up;
   down = floor(number);
   up = ceil(number);
   printf("original number     %5.2lf\n", number);
   printf("number rounded down %5.2lf\n", down);
   printf("number rounded up   %5.2lf\n", up);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ceil	+	+	+	+
ceill		+	+	+

3.1.4.19.16 cos, cosl

Header File

math.h

Category

Math Routines, Inline Routines

Prototype

```
double cos(double x);
```

```
long double cosl(long double x);
```

Description

Calculates the cosine of a value.

cos computes the cosine of the input value. The angle is specified in radians.

cosl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

cos of a real argument returns a value in the range -1 to 1. Error handling for these functions can be modified through _matherr (or _matherrl).

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 0.5;
   result = cos(x);
   printf("The cosine of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
cos	+	+	+	+
cosl		+	+	+

3.1.4.19.17 cosh, coshl

Header File

math.h

Category

Math Routines

Prototype

```
double cosh(double x);
```

```
long double coshl(long double x);
```

Description

Calculates the hyperbolic cosine of a value.

cosh computes the hyperbolic cosine:

coshl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

cosh returns the hyperbolic cosine of the argument.

When the correct value would create an overflow, these functions return the value HUGE_VAL (cosh) or _LHUGE_VAL (coshl) with the appropriate sign, and the global variable errno is set to ERANGE. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 0.5;
   result = cosh(x);
   printf("The hyperbolic cosine of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
cosh	+	+	+	+
coshl		+	+	+

3.1.4.19.18 exp, expl

Header File

math.h

Category

Math Routines

Prototype

```
double exp(double x);
```

```
long double expl(long double x);
```

Description

Calculates the exponential e to the x.

expl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

exp returns e to the x.

Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value

overflows, exp returns the value HUGE_VAL and expl returns _LHUGE_VAL. Results of excessively large magnitude cause the global variable errno to be set to

ERANGE	Result out of range

On underflow, these functions return 0.0, and the global variable errno is not changed. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result;
   double x = 4.0;
   result = exp(x);
   printf("'e' raised to the power \
          of %lf (e ^ %lf) = %lf\n",
          x, x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
exp	+	+	+	+
expl		+	+	+

3.1.4.19.19 fabs, fabsl

Header File

math.h

Category

Math Routines

Prototype

```
double fabs(double x);
```

```
long double fabsl(long double x);
```

Description

Returns the absolute value of a floating-point number.

fabs calculates the absolute value of x, a double. fabsl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

fabs and fabsl return the absolute value of x.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
```

```
    float  number = -1234.0;

    printf("number: %f  absolute value: %f\n", number, fabs(number));
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fabs	+	+	+	+
fabsl		+	+	+

3.1.4.19.20 floor, floorl

Header File

math.h

Category

Math Routines

Prototype

```
double floor(double x);
```

```
long double floorl(long double x);
```

Description

Rounds down.

floor finds the largest integer not greater than x.

floorl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

floor returns the integer found as a **double**. floorl returns the integer found as a **long double**.

Portability

	POSIX	Win32	ANSI C	ANSI C++
floor	+	+	+	+
floorl		+	+	+

3.1.4.19.21 fmod, fmodl

Header File

math.h

Category

Math Routines

Prototype

```
double fmod(double x, double y);
```

```
long double fmodl(long double x, long double y);
```

Description

Calculates x modulo y, the remainder of x/y.

fmod calculates x modulo y (the remainder f, where x = ay + f for some integer a, and 0 < f < y).

fmodl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

fmod and fmodl return the remainder f where x = ay + f (as described above). When y = 0, fmod and fmodl return 0.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double x = 5.0, y = 2.0;
   double result;
   result = fmod(x,y);
   printf("The remainder of (%lf / %lf) is %lf\n", x, y, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fmod	+	+	+	+
fmodl		+	+	+

3.1.4.19.22 frexp, frexpl

Header File

math.h

Category

Math Routines

Prototype

```
double frexp(double x, int *exponent);
```

```
long double frexpl(long double x, int *exponent);
```

Description

Splits a number into mantissa and exponent.

frexp calculates the mantissa m (a **double** greater than or equal to 0.5 and less than 1) and the integer value n such that x (the original **double** value) equals m * 2n. frexp stores n in the integer that exponent points to.

frexpl is the **long double** version; it takes a **long double** argument for x and returns a **long double** result.

Return Value

frexp and frexpl return the mantissa m. Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double mantissa, number;
   int exponent;
   number = 8.0;
   mantissa = frexp(number, &exponent);
   printf("The number %lf is ", number);
   printf("%lf times two to the ", mantissa);
   printf("power of %d\n", exponent);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
frexp	+	+	+	+
frexpl		+	+	+

3.1.4.19.23 hypot, hypotl

Header File

math.h

Category

Math Routines

Prototype

```
double hypot(double x, double y);
```

```
long double hypotl(long double x, long double y);
```

Description

Calculates hypotenuse of a right triangle.

hypot calculates the value z where

z2 = x2 + y2 and z >= 0

This is equivalent to the length of the hypotenuse of a right triangle, if the lengths of the two sides are x and y.

hypotl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

On success, these functions return z, a **double** (hypot) or a **long double**) (hypotl). On error (such as an overflow), they set the global variable errno to

ERANGE	Result out of range

and return the value HUGE_VAL (hypot) or _LHUGE_VAL) (hypotl). Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
```

```
int main(void)
{
 double result;
 double x = 3.0;
    double y = 4.0;
    result = hypot(x, y);
    printf("The hypotenuse is: %lf\n", result);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
hypot		+	+	+
hypotl		+	+	+

3.1.4.19.24 ldexp, ldexpl

Header File

math.h

Category

Math Routines

Prototype

```
double ldexp(double x, int exp);
```

```
long double ldexpl(long double x, int exp);
```

Description

Calculates x * 2^exp.

lexpl is the **long double** version; it takes a **long double** argument for x and returns a **long double** result.

Return Value

On success, ldexp (or ldexpl) returns the value it calculated, x * 2^exp. Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double value;
    double x = 2;
    /* ldexp raises 2 by a power of 3
       then multiplies the result by 2   */
    value = ldexp(x,3);
    printf("The ldexp value is: %lf\n", value);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ldexp	+	+	+	+

ldexpl		+	+	+

3.1.4.19.25 ldiv

Header File

math.h

Category

Math Routines

Prototype

```
ldiv_t ldiv(long int numer, long int denom);
```

Description

Divides two **longs**, returning quotient and remainder.

ldiv divides two **longs** and returns both the quotient and the remainder as an ldiv_t type. numer and denom are the numerator and denominator, respectively.

The ldiv_t type is a structure of **long**s defined in stdlib.h as follows:

```
typedef struct {
long int quot; /* quotient */
long int rem; /* remainder */
} ldiv_t;
```

Return Value

ldiv returns a structure whose elements are quot (the quotient) and rem (the remainder).

Example

```
/* ldiv example */
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   ldiv_t lx;
   lx = ldiv(100000L, 30000L);
  printf("100000 div 30000 = %ld remainder %ld\n", lx.quot, lx.rem);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.19.26 log, logl

Header File

math.h

Category

Math Routines

Prototype

```
double log(double x);
long double logl(long double x);
```

Description

Calculates the natural logarithm of x.

log calculates the natural logarithm of x.

logl is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

On success, log and logl return the value calculated, ln(x).

errno

EDOM	Domain error

If x is 0, the functions return the value negative HUGE_VAL (log) or negative _LHUGE_VAL (logl), and set errno to ERANGE. Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double result;
   double x = 8.6872;
   result = log(x);
   printf("The natural log of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
log	+	+	+	+
logl		+	+	+

3.1.4.19.27 log10, log10l

Header File

math.h

Category

Math Routines

Prototype

```
double log10(double x);
long double log10l(long double x);
```

Description

log10 calculates the base ten logarithm of x.

log10l is the **long double** version; it takes a **long double** argument and returns a **long double** result.

Return Value

On success, log10 (or log10l) returns the calculated value log base ten of x.

If the argument x passed to these functions is real and less than 0, the global variable errno is set to

EDOM	Domain error

If x is 0, these functions return the value negative HUGE_VAL (log10) or _LHUGE_VAL (log10l). Error handling for these routines can be modified through the functions _matherr and _matherrl.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double result;
   double x = 800.6872;
   result = log10(x);
   printf("The common log of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
log10	+	+	+	+
log10l		+	+	+

3.1.4.19.28 modf, modfl

Header File

math.h

Category

Math Routines

Prototype

```
double modf(double x, double *ipart);
long double modfl(long double x, long double *ipart);
```

Description

Splits a **double** or **long double** into integer and fractional parts.

modf breaks the **double** x into two parts: the integer and the fraction. modf stores the integer in ipart and returns the fraction.

modfl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

modf and modfl return the fractional part of x.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
    double fraction, integer;
    double number = 100000.567;
    fraction = modf(number, &integer);
    printf("The whole and fractional parts of %lf are %lf and %lf\n",
           number, integer, fraction);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
modf	+	+	+	+
modfl		+	+	+

3.1.4.19.29 poly, polyl

Header File

math.h

Category

Math Routines

Prototype

```
double poly(double x, int degree, double coeffs[]);
```

```
long double polyl(long double x, int degree, long double coeffs[]);
```

Description

Generates a polynomial from arguments.

poly generates a polynomial in x, of degree degree, with coefficients coeffs[0], coeffs[1], ..., coeffs[degree]. For example, if n = 4, the generated polynomial is:

polyl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

poly and polyl return the value of the polynomial as evaluated for the given x.

Example

```
#include <stdio.h>
#include <math.h>
/* polynomial:  x**3 - 2x**2 + 5x - 1 */
int main(void)
{
  double array[] = { -1.0, 5.0, -2.0, 1.0
};
  double result;
  result = poly(2.0, 3, array);
  printf("The polynomial: x**3 - 2.0x**2 + 5x - 1 at 2.0 is %lf\n", result);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
poly		+		
polyl		+		

3.1.4.19.30 pow, powl

Header File

math.h

Category

Math Routines

Prototype

```
double pow(double x, double y);
long double powl(long double x, long double y);
```

Description

Calculates x to the power of y.

powl is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

On success, pow and powl return the value calculated of x to the power of y.

Sometimes the arguments passed to these functions produce results that overflow or are incalculable. When the correct value would overflow, the functions return the value HUGE_VAL (pow) or _LHUGE_VAL (powl). Results of excessively large magnitude can cause the global variable errno to be set to

ERANGE	Result out of range

If the argument x passed to pow or powl is real and less than 0, and y is not a whole number, or if x is 0 and y is less than 0, or you call pow(0,0), the global variable errno is set to

EDOM	Domain error

Error handling for these functions can be modified through the functions _matherr and _matherrl.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
  double x = 2.0, y = 3.0;
  printf("%lf raised to %lf is %lf\n", x, y, pow(x, y));
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
pow	+	+	+	+

powl		+	+	+

3.1.4.19.31 pow10, pow10l

Header File

math.h

Category

Math Routines

Prototype

```
double pow10(int p);
long double pow10l(int p);
```

Description

Calculates 10 to the power of p.

pow10l is the **long double** version; it takes **long double** arguments and returns a **long double** result.

Return Value

On success, pow10 returns the value calculated, 10 to the power of p and pow10l returns a **long double** result.

The result is actually calculated to **long double** accuracy. All arguments are valid, although some can cause an underflow or overflow.

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
  double p = 3.0;
  printf("Ten raised to %lf is %lf\n", p, pow10(p));
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
pow10		+		
pow10l		+		

3.1.4.19.32 sin, sinl

Header File

math.h

Category

Math Routines

Prototype

```
double sin(double x);
long double sinl(long double x);
```

Description

Calculates sine.

sin computes the sine of the input value. Angles are specified in radians.

sinl is the **long double** version; it takes a l**ong double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Return Value

sin and sinl return the sine of the input value.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result, x = 0.5;
   result = sin(x);
   printf("The sin of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sin	+	+	+	+
sinl		+	+	+

3.1.4.19.33 sinh, sinhl

Header File

math.h

Category

Math Routines, Inline Routines

Prototype

```
double sinh(double x);
long double sinhl(long double x);
```

Description

Calculates hyperbolic sine.

sinh computes the hyperbolic sine.

sinl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for sinh and sinhl can be modified through the functions _matherr and _matherrl.

Return Value

sinh and sinhl return the hyperbolic sine of x.

When the correct value overflows, these functions return the value HUGE_VAL (sinh) or _LHUGE_VAL (sinhl) of appropriate sign. Also, the global variable errno is set to ERANGE.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result, x = 0.5;
   result = sinh(x);
   printf("The hyperbolic sin of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sinh	+	+	+	+
sinhl		+	+	+

3.1.4.19.34 sqrt, sqrtl

Header File

math.h

Category

Math Routines

Prototype

```
double sqrt(double x);
```

```
long double sqrtl(long double x);
```

Description

Calculates the positive square root.

sqrt calculates the positive square root of the argument x.

sqrtl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Return Value

On success, sqrt and sqrtl return the value calculated, the square root of x. If x is real and positive, the result is positive. If x is real and negative, the global variable errno is set to

EDOM	Domain error

Example

```
#include <math.h>
#include <stdio.h>
int main(void)
{
   double x = 4.0, result;
   result = sqrt(x);
   printf("The square root of %lf is %lf\n", x, result);
```

```
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sqrt	+	+	+	+
sqrtl		+	+	+

3.1.4.19.35 tan, tanl

Header File

math.h

Category

Math Routines

Prototype

```
double tan(double x);
```

```
long double tanl(long double x);
```

Description

Calculates the tangent.

tan calculates the tangent. Angles are specified in radians.

tanl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these routines can be modified through the functions _matherr and _matherrl..

Return Value

tan and tanl return the tangent of x, sin(x)/cos(x).

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result, x;
   x = 0.5;
   result = tan(x);
   printf("The tan of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
tan	+	+	+	+
tanl		+	+	+

3.1.4.19.36 tanh, tanhl

Header File

math.h

Category

Math Routines

Prototype

```
double tanh(double x);
```

```
long double tanhl(long double x);
```

Description

Calculates the hyperbolic tangent.

tanh computes the hyperbolic tangent, sinh(x)/cosh(x).

tanhl is the **long double** version; it takes a **long double** argument and returns a **long double** result. Error handling for these functions can be modified through the functions _matherr and _matherrl.

Return Value

tanh and tanhl return the hyperbolic tangent of x.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   double result, x;
   x = 0.5;
   result = tanh(x);
   printf("The hyperbolic tangent of %lf is %lf\n", x, result);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
tanh	+	+	+	+
tanhl		+	+	+

3.1.4.20 mem.h

The following functions, macros, and classes are provided in `mem.h`:

Topics

Name	Description
memccpy (see page 962)	**Header File** mem.h, string.h **Category** Memory and String Manipulation Routines **Prototype** `void *memccpy(void *dest, const void *src, int c, size_t n);` **Description** Copies a block of n bytes. memccpy is available on UNIX System V systems. memccpy copies a block of n bytes from src to dest. The copying stops as soon as either of the following occurs: • The character c is first copied into dest. • n bytes have been copied into dest. **Return Value** memccpy returns a pointer to the byte in dest immediately following c, if c was copied; otherwise, memccpy returns NULL. **Example**

memchr, _wmemchr (see page 963)	**Header File**
	mem.h, string.h
	Category
	Memory and String Manipulation Routines, Inline Routines, C++ Prototyped Routines
	Prototype
	`void *memchr(const void s, int c, size_t n);/ C only */`
	`const void *memchr(const void *s, int c, size_t n);// C++ only`
	`void *memchr(void *s, int c, size_t n);// C++ only`
	`void *memchr(const void s, int c, size_t n);/ C and C++ */`
	`void * _wmemchr(void s, int c, size_t n);/ unicode version */`
	Description
	Searches n bytes for character c.
	memchr is available on UNIX System V systems.
	memchr searches the first n bytes of the block pointed to by s for... more (see page 963)

memcmp (see page 964)	**Header File** mem.h, string.h **Category** Memory and String Manipulation Routines, Inline Routines **Prototype** `int memcmp(const void *s1, const void *s2, size_t n);` **Description** Compares two blocks for a length of exactly n bytes. memcmp is available on UNIX System V systems. memcmp compares the first n bytes of the blocks s1 and s2 as **unsigned char**s. **Return Value** Because it compares bytes as **unsigned char**s, memcmp returns a value that is • < 0 if s1 is less than s2 • = 0 if s1 is the same as s2 • > 0 if s1 is greater than s2 For... more (see page 964)

memcpy, _wmemcpy (see page 965)	**Header File**
	mem.h, string.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`void *memcpy(void *dest, const void *src, size_t n);`
	`void *_wmemcpy(void *dest, const void *src, size_t n);`
	Description
	Copies a block of n bytes.
	memcpy is available on UNIX System V systems.
	memcpy copies a block of n bytes from src to dest. If src and dest overlap, the behavior of memcpy is undefined.
	Return Value
	memcpy returns dest.
	Example

memicmp (see page 965)	**Header File** mem.h, string.h **Category** Memory and String Manipulation Routines **Prototype** `int memicmp(const void *s1, const void *s2, size_t n);` **Description** Compares n bytes of two character arrays, ignoring case. memicmp is available on UNIX System V systems. memicmp compares the first n bytes of the blocks s1 and s2, ignoring character case (upper or lower). **Return Value** memicmp returns a value that is • < 0 if s1 is less than s2 • = 0 if s1 is the same as s2 • > 0 if s1 is greater than s2 **Example**
memmove (see page 966)	**Header File** mem.h, string.h **Category** Memory and String Manipulation Routines **Prototype** `void *memmove(void *dest, const void *src, size_t n);` **Description** Copies a block of n bytes. memmove copies a block of n bytes from src to dest. Even when the source and destination blocks overlap, bytes in the overlapping locations are copied correctly. **Return Value** memmove returns dest. **Example**

memset, _wmemset (see page 967)	**Header File**
	mem.h, string.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`void *memset(void *s, int c, size_t n);`
	`void *_wmemset(void *s, int c, size_t n);`
	Description
	Sets n bytes of a block of memory to byte c.
	memset sets the first n bytes of the array s to the character c.
	Return Value
	memset returns s.
	Example

setmem (see page 968)	**Header File** mem.h **Category** Memory and String Manipulation Routines **Prototype** `void setmem(void *dest, unsigned length, char value);` **Description** Assigns a value to a range of memory. setmem sets a block of length bytes, pointed to by dest, to the byte value. **Return Value** None. **Example**

3.1.4.20.1 memccpy

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines

Prototype

```
void *memccpy(void *dest, const void *src, int c, size_t n);
```

Description

Copies a block of n bytes.

memccpy is available on UNIX System V systems.

memccpy copies a block of n bytes from src to dest. The copying stops as soon as either of the following occurs:

- The character c is first copied into dest.
- n bytes have been copied into dest.

Return Value

memccpy returns a pointer to the byte in dest immediately following c, if c was copied; otherwise, memccpy returns NULL.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *src = "This is the source string";
   char dest[50];
   char *ptr;
   ptr = (char *) memccpy(dest, src, 'c', strlen(src));
   if (ptr)
   {
      *ptr = '\0';
      printf("The character was found: %s\n", dest);
   }
   else
      printf("The character wasn't found\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.20.2 memchr, _wmemchr

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines, Inline Routines, C++ Prototyped Routines

Prototype

```
void *memchr(const void s, int c, size_t n);/ C only */
const void *memchr(const void *s, int c, size_t n);// C++ only
void *memchr(void *s, int c, size_t n);// C++ only
void *memchr(const void s, int c, size_t n);/ C and C++ */
void * _wmemchr(void s, int c, size_t n);/ unicode version */
```

Description

Searches n bytes for character c.

memchr is available on UNIX System V systems.

memchr searches the first n bytes of the block pointed to by s for character c.

Return Value

On success, memchr returns a pointer to the first occurrence of c in s; otherwise, it returns NULL.

Note: If you are using the intrinsic version of these functions, the case of n = 0 will return NULL.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
    char str[17];
    char *ptr;
    strcpy(str, "This is a string");
    ptr = (char *) memchr(str, 'r', strlen(str));
    if (ptr)
       printf("The character 'r' is at position: %d\n", ptr - str);
    else
       printf("The character was not found\n");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
memchr	+	+	+	+
_wmemchr		+		

3.1.4.20.3 memcmp

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

Compares two blocks for a length of exactly n bytes.

memcmp is available on UNIX System V systems.

memcmp compares the first n bytes of the blocks s1 and s2 as **unsigned char**s.

Return Value

Because it compares bytes as **unsigned char**s, memcmp returns a value that is

- < 0 if s1 is less than s2
- = 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

For example,

```
memcmp(“\xFF”, “\x7F”, 1)
```

returns a value greater than 0.

Note: If you are using the intrinsic version of these functions, the case of n = 0 will return NULL.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *buf1 = "aaa";
   char *buf2 = "bbb";
   char *buf3 = "ccc";
   int stat;
   stat = memcmp(buf2, buf1, strlen(buf2));
   if (stat > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");
   stat = memcmp(buf2, buf3, strlen(buf2));
   if (stat > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.20.4 memcpy, _wmemcpy

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
void *memcpy(void *dest, const void *src, size_t n);
```

```
void *_wmemcpy(void *dest, const void *src, size_t n);
```

Description

Copies a block of n bytes.

memcpy is available on UNIX System V systems.

memcpy copies a block of n bytes from src to dest. If src and dest overlap, the behavior of memcpy is undefined.

Return Value

memcpy returns dest.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char src[] = "******************************";
   char dest[] = "abcdefghijlkmnopqrstuvwxyz0123456709";
   char *ptr;
   printf("destination before memcpy: %s\n", dest);
   ptr = (char *) memcpy(dest, src, strlen(src));
   if (ptr)
      printf("destination after memcpy: %s\n", dest);
   else
      printf("memcpy failed\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
memcpy	+	+	+	+
_wmemcpy		+		

3.1.4.20.5 memicmp

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines

Prototype

```
int memicmp(const void *s1, const void *s2, size_t n);
```

Description

Compares n bytes of two character arrays, ignoring case.

memicmp is available on UNIX System V systems.

memicmp compares the first n bytes of the blocks s1 and s2, ignoring character case (upper or lower).

Return Value

memicmp returns a value that is

- < 0 if s1 is less than s2
- = 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *buf1 = "ABCDE123";
   char *buf2 = "abcde456";
   int stat;
   stat = memicmp(buf1, buf2, 5);
   printf("The strings to position 5 are ");
   if (stat)
      printf("not ");
   printf("the same\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.20.6 memmove

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines

Prototype

```
void *memmove(void *dest, const void *src, size_t n);
```

Description

Copies a block of n bytes.

memmove copies a block of n bytes from src to dest. Even when the source and destination blocks overlap, bytes in the overlapping locations are copied correctly.

Return Value

memmove returns dest.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
  char *dest = "abcdefghijklmnopqrstuvwxyz0123456789";
  char *src = "******************************";
  printf("destination prior to memmove: %s\n", dest);
  memmove(dest, src, 26);
  printf("destination after memmove:    %s\n", dest);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
memmove	+	+	+	+
_fmemmove				

3.1.4.20.7 memset, _wmemset

Header File

mem.h, string.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
void *memset(void *s, int c, size_t n);
```

```
void *_wmemset(void *s, int c, size_t n);
```

Description

Sets n bytes of a block of memory to byte c.

memset sets the first n bytes of the array s to the character c.

Return Value

memset returns s.

Example

```
#include <string.h>
#include <stdio.h>
#include <mem.h>
int main(void)
{
    char buffer[] = "Hello world\n";
    printf("Buffer before memset: %s\n", buffer);
    memset(buffer, '*', strlen(buffer) - 1);
    printf("Buffer after memset:  %s\n", buffer);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
memset	+	+	+	+
_wmemset		+		

3.1.4.20.8 setmem

Header File

mem.h

Category

Memory and String Manipulation Routines

Prototype

```
void setmem(void *dest, unsigned length, char value);
```

Description

Assigns a value to a range of memory.

setmem sets a block of length bytes, pointed to by dest, to the byte value.

Return Value

None.

Example

```
#include <stdio.h>
#include <alloc.h>
#include <mem.h>
int main(void)
{
   char *dest;
   dest = (char *) calloc(21, sizeof(char));
   setmem(dest, 20, 'c');
   printf("%s\n", dest);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.21 new.h

The following functions, macros, and classes are provided in `new.h`:

Topics

Name	Description

_new_handler (see page 969)	**Header File**
	new.h
	Syntax
	`typedef void (*pvf)();`
	`pvf _new_handler;`
	Description
	_new_handler contains a pointer to a function that takes no arguments and returns **void**. If **operator new()** is unable to allocate the space required, it will call the function pointed to by _new_handler; if that function returns it will try the allocation again. By default, the function pointed to by _new_handler simply terminates the application. The application can replace this handler, however, with a function that can try to free up some space. This is done by assigning directly to _new_handler or by calling the function set_new_handler, which returns... more (see page 969)

set_new_handler function (see page 970)	**Header File**
	new.h
	Category
	Memory Routines
	Syntax
	`typedef void (new * new_handler)();`
	`new_handler set_new_handler(new_handler my_handler);`
	Description
	set_new_handler installs the function to be called when the global **operator new** or **operator new[]()** cannot allocate the requested memory. By default the new operators throw an bad_alloc exception if memory cannot be allocated. You can change this default behavior by calling set_new_handler to set a new handler. To retain the traditional version of new, which does not throw exceptions, you can use set_new_handler(0).
	If new cannot allocate the requested memory, it calls the handler that was set by a previous call to set_new_handler.... more (see page 970)

3.1.4.21.1 _new_handler

Header File

new.h

Syntax

```
typedef void (*pvf)();
pvf _new_handler;
```

Description

_new_handler contains a pointer to a function that takes no arguments and returns **void**. If **operator new()** is unable to allocate the space required, it will call the function pointed to by _new_handler; if that function returns it will try the allocation again. By default, the function pointed to by _new_handler simply terminates the application. The application can replace this handler, however, with a function that can try to free up some space. This is done by assigning directly to _new_handler or by calling the function set_new_handler, which returns a pointer to the former handler.

As an alternative, you can set using the function set_new_handler, like this:

```
pvf set_new_handler(pvf p);
```

_new_handler is provided primarily for compatibility with C++ version 1.2. In most cases this functionality can be better provided by overloading **operator new()**.

Example

3.1.4.21.2 set_new_handler function

Header File

new.h

Category

Memory Routines

Syntax

```
typedef void (new * new_handler)();
new_handler set_new_handler(new_handler my_handler);
```

Description

set_new_handler installs the function to be called when the global **operator new** or **operator new[]()** cannot allocate the requested memory. By default the new operators throw an bad_alloc exception if memory cannot be allocated. You can change this default behavior by calling set_new_handler to set a new handler. To retain the traditional version of new, which does not throw exceptions, you can use set_new_handler(0).

If new cannot allocate the requested memory, it calls the handler that was set by a previous call to set_new_handler. If there is no handler installed by set_new_handler, new returns 0. my_handler should specify the actions to be taken when new cannot satisfy a request for memory allocation. The new_handler type, defined in new.h, is a function that takes no arguments and returns **void**. A new_handler can throw a bad_alloc exception.

- The user-defined my_handler should do one of the following:
- return after freeing memory
- throw an bad_alloc exception or an exception derived from bad_alloc
- call abort or exit functions

If my_handler returns, then new will again attempt to satisfy the request.

Ideally, my_handler would free up memory and return. new would then be able to satisfy the request and the program would continue. However, if my_handler cannot provide memory for new, my_handler must throw an exception or terminate the program. Otherwise, an infinite loop will be created.

Preferably, you should overload **operator new()** and **operator new[]()** to take appropriate actions for your applications.

Return Value

set_new_handler returns the old handler, if one has been registered.

The user-defined argument function, my_handler, should not return a value.

Example

```
#include <iostream>
#include <new.h>
#include <stdlib.h>
using std::cout;
using std::hex;
void mem_warn() {
  std::cerr << "\nCan't allocate!";
  exit(1);
}

void main(void) {
  std::set_new_handler(mem_warn);
  char *ptr = new char[100];
  cout << "\nFirst allocation: ptr = " << hex << long(ptr);
  ptr = new char[64000U];
```

```
  cout << "\nFinal allocation: ptr = " << hex << long(ptr);
  std::set_new_handler(0);  // Reset to default.
}
```

3.1.4.22 process.h

The following functions, macros, and classes are provided in `process.h`:

Topics

Name	Description
P_xxxx #defines (see page 975)	**Header File** process.h **Description** Modes used by the spawn... functions.
_adopt_thread (see page 975)	**Header File** process.h **Category** Process Control Routines **Prototype** `_PTHREAD_ADOPTION_DATA _adopt_thread(void (_USERENTRY *__start_address)(void *), void * __arglist, int free_flag );` **Description** “Adopts” a thread, created with the Windows API CreateThread function, to the C++Builder RTL by hooking up the necessary internal data (exceptions, stack info, and so forth). _adopt_thread thereby allows the RTL to handle exception issues in that thread. The execution path of this thread is then transferred to another function (the adoptive thread function). From the RTL's perspective, during the execution of this adoptive thread function, the thread appears as if it had been created by a call to... more (see page 975)
_beginthread (see page 976)	**Header File** process.h **Category** Process Control Routines **Prototype** `unsigned long _beginthread(void (_USERENTRY *__start)(void *), unsigned __stksize, void *__arg);` **Description** Starts execution of a new thread. **Note:** The start_address must be declared to be _USERENTRY. The _beginthread function creates and starts a new thread. The thread starts execution at start_address. The size of its stack in bytes is stack_size; the stack is allocated by the operating system after the stack size is rounded up to the next multiple of 4096. The thread is passed arglist as its only parameter; it can be NULL, but must be present. The thread function should... more (see page 976)
_beginthreadNT (see page 978)	**Header File** process.h **Category** Process Control Routines **Prototype** `unsigned long _beginthreadNT(void (_USERENTRY *start_address)(void *), unsigned stack_size, void *arglist, void *security_attrib, unsigned long create_flags, unsigned long *thread_id);` **Description** Starts execution of a new thread under Windows NT. **Note:** The start_address must be declared to be _USERENTRY. All multithread Windows NT programs must use _beginthreadNT or the _beginthread function instead of the operating system thread-creation API function because these functions perform initialization required for correct operation of the runtime library functions. The _beginthreadNT function provides support for the operating system security. These functions are available only in the multithread libraries. The _beginthreadNT... more (see page 978)

_beginthreadex (see page 980)	**Header File** process.h **Category** Process Control Routines **Prototype** `unsigned long _beginthreadex(void *__security_attr, unsigned __stksize, unsigned (__stdcall *__start)(void *), void *__arg, unsigned __create_flags, unsigned *__thread_id);` **Description** Creates a thread and allows specifying the other parameters of the OS API CreateThread (such as security and thread creation flags). The _endthreadex function will be called automatically when the thread function terminates. The value returned from your thread function will be passed along to _endthreadex, which in turn will pass it along to the ExitThread API. The return value can then be retrieved using the GetExitCodeThread API. Unlike _endthread, the _endthreadex function does **not... more (see page 980)**
_c_exit (see page 981)	**Header File** process.h **Category** Process Control Routines **Prototype** `void _c_exit(void);` **Description** Performs _exit cleanup without terminating the program. _c_exit performs the same cleanup as _exit, except that it does not terminate the calling process. **Return Value** None. **Example**
_cexit (see page 982)	**Header File** process.h **Category** Process Control Routines **Prototype** `void _cexit(void);` **Description** Performs exit cleanup without terminating the program. _cexit performs the same cleanup as exit, closing all files but without terminating the calling process. The _cexit function calls any registered "exit functions" (posted with atexit). Before _cexit returns, it flushes all input/output buffers and closes all streams that were open. **Return Value** None. **Example**
_endthread (see page 983)	**Header File** process.h **Category** Process Control Routines **Prototype** `void _endthread(void);` **Description** Terminates execution of a thread. The _endthread function terminates the currently executing thread by closing the thread handle and calling the ExitThread API. The thread must have been started by an earlier call to _beginthread or _beginthreadNT.. _endthread is called automatically by the runtime library when your thread function terminates. This function is available in the multithread libraries; it is not in the single-threaded libraries. **Return Value** The function does not return a value.

_endthreadex (see page 984)	**Header File** process.h **Category** Process Control Routines **Prototype** `void _endthreadex(unsigned thread_retval);` **Description** Terminates execution of the current thread by calling the ExitThread API, but without closing the handle. The thread must have been created by an earlier call to _beginthreadex. The runtime library will call _endthreadex autotmatically, when your thread function terminates. _endthreadex receives the return value of your thread function in thread_retval, and will pass it along to the Win32 ExitThread API. **Note:** Note: Performs the same operation as _endthread(), but does not close the thread handle. **Return Value** None.
_expand (see page 984)	**Header File** process.h **Category** Memory Routines **Prototype** `void *_expand(void *block, size_t size);` **Description** Grows or shrinks a heap block in place. This function attempts to change the size of an allocated memory block without moving the block's location in the heap. The data in the block are not changed, up to the smaller of the old and new sizes of the block. The block must have been allocated earlier with malloc, calloc, or realloc, and must not have been freed. **Return Value** If _expand is able to resize the block without moving it, _expand returns a pointer to the block,... more (see page 984)
_unadopt_thread (see page 985)	**Header File** process.h **Category** Process Control Routines **Prototype** `void _unadopt_thread(_PTHREAD_ADOPTION_DATA thd);` **Description** Frees the RTL thread-specific data associated with a previous call to _adopt_thread. **Return Value** None. **Portability**
cwait (see page 985)	**Header File** process.h **Category** Process Control Routines **Prototype** `int cwait(int *statloc, int pid, int action);` **Description** Waits for child process to terminate. The cwait function waits for a child process to terminate. The process ID of the child to wait for is pid. If statloc is not NULL, it points to the location where cwait will store the termination status. The action specifies whether to wait for the process alone, or for the process and all of its children. If the child process terminated normally (by calling exit, or returning from main), the termination status word is defined as follows:... more (see page 985)

execl, execle, execlp, execlpe, execv, execve, execvp, execvpe, _wexecl, _wexecle, _wexeclp, _wexeclpe, _wexecv, _wexecve, _wexecvp, _wexecvpe (see page 987)	**Header File**
	process.h
	Category
	Process Control Routines
	Prototype
	`int execl(char *path, char *arg0 *arg1, ..., *argn, NULL);`
	`int _wexecl(wchar_t *path, wchar_t *arg0 *arg1, ..., *argn, NULL);`
	`int execle(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);`
	`int _wexecle(wchar_t *path, wchar_t *arg0, *arg1, ..., *argn, NULL, wchar_t **env);`
	`int execlp(char *path, char *arg0,*arg1, ..., *argn, NULL);`
	`int _wexeclp(wchar_t *path, wchar_t *arg0,*arg1, ..., *argn, NULL);`
	`int execlpe(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);`
	`int _wexeclpe(wchar_t *path, wchar_t *arg0, *arg1, ..., *argn, NULL, wchar_t **env);`
	`int execv(char *path, char *argv[]);`
	`int _wexecv(wchar_t *path, wchar_t *argv[]);`
	`int execve(char *path, char... more (see page 987)`

getpid (see page 989)	**Header File** process.h **Category** Process Control Routines **Prototype** unsigned getpid(void) **Description** Gets the process ID of a program. This function returns the current process ID--an integer that uniquely identifies the process. **Return Value** getpid returns the current process' ID. **Example**

spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe, _wspawnl, _wspawnle, _wspawnlp, _wspawnlpe, _wspawnv, _wspawnve, _wspawnvp, _wspawnvpe (see page 989)	**Header File**
	process.h
	Category
	Process Control Routines
	Prototype
	`int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL);`
	`int _wspawnl(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL);`
	`int spawnle(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char *envp[]);`
	`int _wspawnle(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL, wchar_t *envp[]);`
	`int spawnlp(int mode, char *path, char *arg0, arg1, ..., argn, NULL);`
	`int _wspawnlp(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL);`
	`int spawnlpe(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char *envp[]);`
	`int _wspawnlpe(int mode, wchar_t *path, wchar_t *arg0, arg1, ...,... more (see page 989)`

wait (see page 992)	**Header File** process.h **Category** Process Control Routines **Prototype** int wait(int *statloc); **Description** Waits for one or more child processes to terminate. The wait function waits for one or more child processes to terminate. The child processes must be those created by the calling program; wait cannot wait for grandchildren (processes spawned by child processes). If statloc is not NULL, it points to location where wait will store the termination status. If the child process terminated normally (by calling exit, or returning from main), the termination status word is defined as follows:

3.1.4.22.1 P_xxxx #defines

Header File

process.h

Description

Modes used by the spawn... functions.

Constant	Meaning
P_WAIT	Child runs separately, parent waits until exit
P_DETACH	Child and parent run concurrently with child process in background mode
P_NOWAIT	Child and parent run concurrently (Not implemented)
P_NOWAITO	Child and parent run concurrently, but the child process is not saved
P_OVERLAY	Child replaces parent so that parent no longer exists

3.1.4.22.2 _adopt_thread

Header File

process.h

Category

Process Control Routines

Prototype

```
_PTHREAD_ADOPTION_DATA _adopt_thread(void (_USERENTRY *__start_address)(void *), void *__arglist, int free_flag );
```

Description

“Adopts” a thread, created with the Windows API CreateThread function, to the C++Builder RTL by hooking up the necessary internal data (exceptions, stack info, and so forth). _adopt_thread thereby allows the RTL to handle exception issues in that thread. The execution path of this thread is then transferred to another function (the adoptive thread function). From the RTL's perspective, during the execution of this adoptive thread function, the thread appears as if it had been created by a call to _beginthreadex and is allowed all the benefits, such as calling other RTL functions and throwing/catching exceptions.

To create a thread, a user normally calls _beginthread. This hooks up the internal data automatically. _adopt_thread is primarily used in cases in which this internal data is not present. For example, this happens when a user is called from a thread that came

from an outside source, such as ISAPI.

Using _adopt_thread thereby allows C++Builder compiled DLLs to be used from non-C++Builder EXEs. _adopt_thread works by:

- calling the user function (passing in the arglist param)
- unhooking the exception information
- returning a handle to the thread context

This process allows the same function to be used again (without reallocating all that data). At the end of this cycle, the _unadopt_thread function can be called to finally free up the rest of this data.

The last parameter, free_flag, determines whether the thread data structures are freed upon function exit:

- If free_flag is zero, then when the adoptive function exits only its exception handler is un-hooked. The rest of the RTL specific data that had been allocated during the thread's adoption, remains valid. If the same thread then calls _adopt_thread again, the existing data is used, the exception handler is rehooked, and the specified adoptive thread function is called.
- If free_flag is set to non-zero, the thread data structures will be freed before _adopt_thread returns. In this case the returned thread handle will be NULL since its associated date has already been freed.

Return Value

If the __free_flag parameter is **false** (zero), _adopt_thread returns a handle (thread context) that can later be used to free these data structures by passing it to _unadopt_thread().

If the __free_flag parameter to _adopt_thread is non-zero, the thread data is freed before _adopt_thread returns, and the returned handle is NULL.

If an error has occurred, errno is set to:

ENOMEM Not enough memory

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.22.3 _beginthread

Header File

process.h

Category

Process Control Routines

Prototype

```
unsigned long _beginthread(void (_USERENTRY *__start)(void *), unsigned __stksize, void *__arg);
```

Description

Starts execution of a new thread.

Note: The start_address must be declared to be _USERENTRY.

The _beginthread function creates and starts a new thread. The thread starts execution at start_address.

The size of its stack in bytes is stack_size; the stack is allocated by the operating system after the stack size is rounded up to the next multiple of 4096. The thread is passed arglist as its only parameter; it can be NULL, but must be present. The thread function should terminate by simply returning; the _endthread. function will be called automatically. The _endthread function will automatically close the handle, and set the return value of the thread to zero.

Either this function or _beginthreadNT must be used instead of the operating system thread-creation API function because _beginthread and _beginthreadNT perform initialization required for correct operation of the runtime library functions.

This function is available only in the multithread libraries.

Return Value

_beginthread returns the handle of the new thread. The return value is a standard Windows handle that can be used in operating system API's such as SuspendThread and ResumeThread.

On error, the function returns -1, and the global variable errno is set to one of the following values:

EAGAIN	Too many threads
EINVAL	Invalid stack size (i.e. less than 16 bytes, or equal to zero)
ENOMEM	Not enough memory

Also see the description of the Win32 API GetLastError, in the MSDN Library.

Example

```
/*  Use the -tWM  (32-bit multi-threaded target) command-line switch for this example  */
#include <stdio.h>
#include <errno.h>
#include <stddef.h>     /* _threadid variable */
#include <process.h>    /* _beginthread, _endthread */
#include <time.h>       /* time, _ctime */
void thread_code(void *threadno)
{
    time_t t;
    time(&t);
    printf("Executing thread number %d, ID = %d, time = %s\n",
        (int)threadno, _threadid, ctime(&t));

}

void start_thread(int i)
{
    int thread_id;
#if     defined(__WIN32__)
    if ((thread_id = _beginthread(thread_code,4096,(void *)i)) == (unsigned long)-1)
#else
    if ((thread_id = _beginthread(thread_code,4096,(void *)i)) == -1)
#endif
    {
        printf("Unable to create thread %d, errno = %d\n",i,errno);
        return;
    }
    printf("Created thread %d, ID = %ld\n",i,thread_id);
}
int main(void)
{
    int i;
    for (i = 1; i < 20; i++)
        start_thread(i);
    printf("Hit ENTER to exit main thread.\n");
    getchar();
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.22.4 _beginthreadNT

Header File

process.h

Category

Process Control Routines

Prototype

```
unsigned long _beginthreadNT(void (_USERENTRY *start_address)(void *), unsigned stack_size,
void *arglist, void *security_attrib, unsigned long create_flags, unsigned long *thread_id);
```

Description

Starts execution of a new thread under Windows NT.

Note: The start_address must be declared to be _USERENTRY.

All multithread Windows NT programs must use _beginthreadNT or the _beginthread function instead of the operating system thread-creation API function because these functions perform initialization required for correct operation of the runtime library functions. The _beginthreadNT function provides support for the operating system security. These functions are available only in the multithread libraries.

The _beginthreadNT function creates and starts a new thread. The thread starts execution at start_address. When your thread terminates, the _endthread function will be called automatically. _endthread will close the thread handle, and call the ExitThread API.

The size of its stack in bytes is stack_size; the stack is allocated by the operating system after the stack size is rounded up to the next multiple of 4096. The thread arglist can be NULL, but must be present.

The _beginthreadNT function uses the security_attr pointer to access the SECURITY_ATTRIBUTES structure. The structure contains the security attributes for the thread. If security_attr is NULL, the thread is created with default security attributes. The thread handle is not inherited if security_attr is NULL.

_beginthreadNT reads the create_flags variable for flags that provide additional information about the thread creation. This variable can be zero, specifying that the thread will run immediately upon creation. The variable can also be CREATE_SUSPENDED; in which case, the thread will not run until the ResumeThread function is called. ResumeThread is provided by the Win32 API.

_beginthreadNT initializes the thread_id variable with the thread identifier.

Return Value

On success, _beginthreadNT returns the handle of the new thread. The return value is a standard Windows handle that can be used in operating system API's such as SuspendThread and ResumeThread.

On error, it returns -1, and the global variable errno is set to one of the following values:

EAGAIN	Too many threads
EINVAL	Invalid stack size (i.e. less than 16 bytes, or equal to zero)
ENOMEM	Not enough memory

Also see the description of the Win32 API GetLastError, in the MSDN Library.

Example

```
/*  Use the -tWM  (32-bit multi-threaded target) command-line switch for this example  */
```

```
#pragma checkoption -tWM
#include        <windows.h>
#include        <process.h>
#include        <stdio.h>
#define NTHREADS 25
static LONG runningThreads = 0;
static HANDLE doneEvent;
/* This function acts as the 'main' function for each new thread */
static void threadMain(void *arg)
{
  printf("Thread %2d has an ID of %u\n", (int)arg, GetCurrentThreadId());
  /* Use InterlockedDecrement() to modify the global runningThreads in a
 * thread safe manner.  When the count hits 0, signal the main thread if
 * it created an event for us.
 */
  if (InterlockedDecrement(&runningThreads) == 0 && doneEvent)
      SetEvent(doneEvent);
}
int main(void)
{
   HANDLE hThreads[NTHREADS];
   int i;
   DWORD  threadId;
   SECURITY_ATTRIBUTES  sa = {
          sizeof(SECURITY_ATTRIBUTES), /* structure size */
          0,      /* No security descriptor */
          TRUE    /* Thread handle is inheritable */
   };
   /* Create NTHREADS inheritable threads that are initially suspended and that will run
starting at threadMain()*/
   for(i = 0; i < NTHREADS; i++) {
      hThreads[i] = (HANDLE)_beginthreadNT(
         threadMain,   /* Thread starting address */
         4096,         /* Thread stack size */
         (void *)i,    /* Thread start argument */
         &sa,          /* Thread security */
         CREATE_SUSPENDED,  /* Create in suspended state */
         &threadId);   /* Thread ID */
      if(hThreads[i] == INVALID_HANDLE_VALUE) {
         MessageBox(0, "Thread Creation Failed", "Error", MB_OK);
         return  1;
      }
      ++runningThreads;
      printf("Created thread %2d with an ID of %u\n", i, threadId);
   }
   printf("\nPress ENTER to thaw all threads\n\n");
   getchar();
   /* Create the event that will signal when all threads are done */
   doneEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

   /* Resume the suspended threads */
   for(i = 0; i < NTHREADS; i++)
      ResumeThread(hThreads[i]);
   /* Wait for all threads to finish execution, if we can */
   if (doneEvent) {
     WaitForSingleObject(doneEvent, INFINITE);
     CloseHandle(doneEvent);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.22.5 _beginthreadex

Header File

process.h

Category

Process Control Routines

Prototype

```
unsigned long _beginthreadex(void *__security_attr, unsigned __stksize, unsigned (__stdcall *__start)(void *), void *__arg, unsigned __create_flags, unsigned *__thread_id);
```

Description

Creates a thread and allows specifying the other parameters of the OS API CreateThread (such as security and thread creation flags). The _endthreadex function will be called automatically when the thread function terminates. The value returned from your thread function will be passed along to _endthreadex, which in turn will pass it along to the ExitThread API. The return value can then be retrieved using the GetExitCodeThread API.

Unlike _endthread, the _endthreadex function does **not** close the thread handle, thereby allowing other threads to block on this one without fear that the handle will be freed out from under the system.

Other than the order of parameters and the closing of the thread handle, _beginthreadex performs same operation as _beginthreadNT.

Note: Note: The start address needs to be defined to return an unsigned, which is the thread exit code.

Return Value

_beginthreadex returns the handle of the new thread. The return value is a standard Windows handle that can be used in operating system API's such as SuspendThread and ResumeThread.

If unsuccessful, 0 is returned, and errno is set as follows:

EAGAIN	Too many threads
ENOMEM	Not enough memory
EINVAL	Bad stack value (i.e. less than 16 bytes or equal to zero)

Also see the description of the Win32 API GetLastError, in the MSDN Library.

Example

```
//*  Use the -tWM  (32-bit multi-threaded target) command-line switch for this example  */
#pragma checkoption -tWM
#include        <windows.h>
#include        <process.h>
#include        <stdio.h>
#define NTHREADS 25
/* This function acts as the 'main' function for each new thread */
static unsigned __stdcall threadMain(void *arg)
{
   printf("Thread %2d has an ID of %u\n", (int)arg, GetCurrentThreadId());
   return 0;
}
int main(void)
{
   HANDLE hThreads[NTHREADS];
   int i;
```

```
    unsigned threadId;
    SECURITY_ATTRIBUTES  sa = {
           sizeof(SECURITY_ATTRIBUTES), /* structure size */
           0,      /* No security descriptor */
           TRUE    /* Thread handle is inheritable */
    };
    /* Create NTHREADS inheritable threads that are initially suspended and that will run
starting at threadMain()*/
    for(i = 0; i < NTHREADS; i++) {
       hThreads[i] = (HANDLE)_beginthreadex(
          &sa,          /* Thread security */
          4096,         /* Thread stack size */
          threadMain,   /* Thread starting address */
          (void *)i,    /* Thread start argument */
          CREATE_SUSPENDED,  /* Create in suspended state */
          &threadId);   /* Thread ID */
       if(hThreads[i] == INVALID_HANDLE_VALUE) {
          MessageBox(0, "Thread Creation Failed", "Error", MB_OK);
          return  1;
       }
       printf("Created thread %2d with an ID of %u\n", i, threadId);
    }
    printf("\nPress ENTER to thaw all threads\n\n");
    getchar();

/* Resume the suspended threads */
    for(i = 0; i < NTHREADS; i++)
       ResumeThread(hThreads[i]);

/* Wait for the threads to run */
    WaitForMultipleObjects(NTHREADS, hThreads, TRUE, INFINITE);
/* Close all of the thread handles */
    for(i = 0; i < NTHREADS; i++)
       CloseHandle(hThreads[i]);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.22.6 _c_exit

Header File

process.h

Category

Process Control Routines

Prototype

```
void _c_exit(void);
```

Description

Performs _exit cleanup without terminating the program.

_c_exit performs the same cleanup as _exit, except that it does not terminate the calling process.

Return Value

None.

Example

```
#include <process.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
main()
{
  int fd;
  char c;

  if ((fd = open("_c_exit.c",O_RDONLY)) < 0)
  {
    printf("Unable to open _c_exit.c for reading\n");
    return 1;
  }
  if (read(fd,&c,1) != 1)
    printf("Unable to read from open file handle %d before _c_exit\n",fd);
  else
    printf("Successfully read from open file handle %d before _c_exit\n",fd);
  _c_exit();
  if (read(fd,&c,1) != 1)
    printf("Unable to read from open file handle %d after _c_exit\n",fd);
  else
    printf("Successfully read from open file handle %d after _c_exit\n",fd);
 return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.22.7 _cexit

Header File

process.h

Category

Process Control Routines

Prototype

```
void _cexit(void);
```

Description

Performs exit cleanup without terminating the program.

_cexit performs the same cleanup as exit, closing all files but without terminating the calling process. The _cexit function calls any registered "exit functions" (posted with atexit). Before _cexit returns, it flushes all input/output buffers and closes all streams that were open.

Return Value

None.

Example

```
#include <windows.h>
```

```
#include <process.h>
#include <io.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
void exit_func(void)
{
  printf("Exit function called\n\n");
  printf("Close Window to return to program... It will beep if able to read from file");
}
int main(void)
{
  int fd;
  char c;
  if ((fd = open("_cexit.c",O_RDONLY)) < 0)
  {
printf("Unable to open _cexit.c for reading\n");
return 1;
  }
  atexit(exit_func);
  if (read(fd,&c,1) != 1)
printf("Unable to read from open file handle %d before _cexit\n",fd);
  else
printf("Successfully read from open file handle %d before _cexit\n",fd);
  _cexit();
  if (read(fd,&c,1) == 1)
  MessageBeep(0);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.22.8 _endthread

Header File

process.h

Category

Process Control Routines

Prototype

```
void _endthread(void);
```

Description

Terminates execution of a thread.

The _endthread function terminates the currently executing thread by closing the thread handle and calling the ExitThread API. The thread must have been started by an earlier call to _beginthread or _beginthreadNT.. _endthread is called automatically by the runtime library when your thread function terminates.

This function is available in the multithread libraries; it is not in the single-threaded libraries.

Return Value

The function does not return a value.

3.1.4.22.9 _endthreadex

Header File

process.h

Category

Process Control Routines

Prototype

```
void _endthreadex(unsigned thread_retval);
```

Description

Terminates execution of the current thread by calling the ExitThread API, but without closing the handle. The thread must have been created by an earlier call to _beginthreadex. The runtime library will call _endthreadex autotmatically, when your thread function terminates. _endthreadex receives the return value of your thread function in thread_retval, and will pass it along to the Win32 ExitThread API.

Note: Note: Performs the same operation as _endthread(), but does not close the thread handle.

Return Value

None.

3.1.4.22.10 _expand

Header File

process.h

Category

Memory Routines

Prototype

```
void *_expand(void *block, size_t size);
```

Description

Grows or shrinks a heap block in place.

This function attempts to change the size of an allocated memory block without moving the block's location in the heap. The data in the block are not changed, up to the smaller of the old and new sizes of the block. The block must have been allocated earlier with malloc, calloc, or realloc, and must not have been freed.

Return Value

If _expand is able to resize the block without moving it, _expand returns a pointer to the block, whose address is unchanged. If _expand is unsuccessful, it returns a NULL pointer and does not modify or resize the block.

Example

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
void main(void)
{
  char *bufchar, *newbuf;
```

```
  printf( "Allocate a 512 element buffer\n" );
  if( (bufchar = (char *) calloc(512, sizeof( char ) )) == NULL)
    exit( 1 );
  printf( "Allocated %d bytes at %Fp\n",
    _msize ( bufchar ), (void __far *)bufchar );
  if ((newbuf = (char *) _expand (bufchar, 1024)) == NULL)
    printf ("cannot expand");
  else {
    bufchar = newbuf;
    printf (" Expanded block to %d bytes at %Fp\n",
      _msize( bufchar ) , (void __far *)bufchar );
  }
  free( bufchar );
  exit (0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.22.11 _unadopt_thread

Header File

process.h

Category

Process Control Routines

Prototype

```
void _unadopt_thread(_PTHREAD_ADOPTION_DATA thd);
```

Description

Frees the RTL thread-specific data associated with a previous call to _adopt_thread.

Return Value

None.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.22.12 cwait

Header File

process.h

Category

Process Control Routines

Prototype

```
int cwait(int *statloc, int pid, int action);
```

Description

Waits for child process to terminate.

The cwait function waits for a child process to terminate. The process ID of the child to wait for is pid. If statloc is not NULL, it points to the location where cwait will store the termination status. The action specifies whether to wait for the process alone, or for the process and all of its children.

If the child process terminated normally (by calling exit, or returning from main), the termination status word is defined as follows:

Bits 0-7	Zero
Bits 8-15	The least significant byte of the return code from the child process. This is the value that is passed to exit, or is returned from main. If the child process simply exited from main without returning a value, this value will be unpredictable.

If the child process terminated abnormally, the termination status word is defined as follows:

Bits 0-7Termination information about the child:

	1	Critical error abort.
	2	Execution fault, protection exception.
	3	External termination signal.
Bits 8-15	Zero	

Bits 8-15	Zero

If pid is 0, cwait waits for any child process to terminate. Otherwise, pid specifies the process ID of the process to wait for; this value must have been obtained by an earlier call to an asynchronous spawn function.

The acceptable values for action are WAIT_CHILD, which waits for the specified child only, and WAIT_GRANDCHILD, which waits for the specified child and all of its children. These two values are defined in process.h.

Return Value

When cwait returns after a normal child process termination, it returns the process ID of the child.

When cwait returns after an abnormal child termination, it returns -1 to the parent and sets errno to EINTR (the child process terminated abnormally).

If cwait returns without a child process completion, it returns a -1 value and sets errno to one of the following values:

ECHILD	No child exists or the pid value is bad
EINVAL	A bad action value was specified

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.22.13 execl, execle, execlp, execlpe, execv, execve, execvp, execvpe, _wexecl, _wexecle, _wexeclp, _wexeclpe, _wexecv, _wexecve, _wexecvp, _wexecvpe

Header File

process.h

Category

Process Control Routines

Prototype

```
int execl(char *path, char *arg0 *arg1, ..., *argn, NULL);
int _wexecl(wchar_t *path, wchar_t *arg0 *arg1, ..., *argn, NULL);
int execle(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);
int _wexecle(wchar_t *path, wchar_t *arg0, *arg1, ..., *argn, NULL, wchar_t **env);
int execlp(char *path, char *arg0,*arg1, ..., *argn, NULL);
int _wexeclp(wchar_t *path, wchar_t *arg0,*arg1, ..., *argn, NULL);
int execlpe(char *path, char *arg0, *arg1, ..., *argn, NULL, char **env);
int _wexeclpe(wchar_t *path, wchar_t *arg0, *arg1, ..., *argn, NULL, wchar_t **env);
int execv(char *path, char *argv[]);
int _wexecv(wchar_t *path, wchar_t *argv[]);
int execve(char *path, char *argv[], char **env);
int _wexecve(wchar_t *path, wchar_t *argv[], wchar_t **env);
int execvp(char *path, char *argv[]);
int _wexecvp(wchar_t *path, wchar_t *argv[]);
int execvpe(char *path, char *argv[], char **env);
int _wexecvpe(wchar_t *path, wchar_t *argv[], wchar_t **env);
```

Description

Loads and runs other programs.

The functions in the exec... family load and run (execute) other programs, known as child processes. When an exec... call succeeds, the child process overlays the parent process. There must be sufficient memory available for loading and executing the child process.

path is the file name of the called child process. The exec... functions search for path using the standard search algorithm:

- If no explicit extension is given, the functions search for the file as given. If the file is not found, they add .EXE and search again. If not found, they add .COM and search again. If found, the command processor, COMSPEC (Windows) or COMMAND.COM (DOS), is used to run the batch file.
- If an explicit extension or a period is given, the functions search for the file exactly as given.

The suffixes l, v, p, and e added to the exec... "family name" specify that the named function operates with certain capabilities.

l	specifies that the argument pointers (arg0, arg1, ..., argn) are passed as separate arguments. Typically, the l suffix is used when you know in advance the number of arguments to be passed.
v	specifies that the argument pointers (argv[0] ..., arg[n]) are passed as an array of pointers. Typically, the v suffix is used when a variable number of arguments is to be passed.
p	specifies that the function searches for the file in those directories specified by the PATH environment variable (without the p suffix, the function searches only the current working directory). If the path parameter does not contain an explicit directory, the function searches first the current directory, then the directories set with the PATH environment variable.
e	specifies that the argument env can be passed to the child process, letting you alter the environment for the child process. Without the e suffix, child processes inherit the environment of the parent process.

Each function in the exec... family must have one of the two argument-specifying suffixes (either l or v). The path search and environment inheritance suffixes (p and e) are optional; for example:

- execl is an exec... function that takes separate arguments, searches only the root or current directory for the child, and passes on the parent's environment to the child.
- execvpe is an exec... function that takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the env argument for altering the child's environment.

The exec... functions must pass at least one argument to the child process (arg0 or argv[0]); this argument is, by convention, a copy of path. (Using a different value for this 0th argument won't produce an error.)

path is available for the child process.

When the l suffix is used, arg0 usually points to path, and arg1, ..., argn point to character strings that form the new list of arguments. A mandatory null following argn marks the end of the list.

When the e suffix is used, you pass a list of new environment settings through the argument env. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

```
envvar = value
```

where envvar is the name of an environment variable, and value is the string value to which envvar is set. The last element in env is null. When env is null, the child inherits the parents' environment settings.

The combined length of arg0 + arg1 + ... + argn (or of argv[0] + argv[1] + ... + argn[n]), including space characters that separate the arguments, must be less than 260 bytes. Null terminators are not counted.

When an exec... function call is made, any open files remain open in the child process.

Return Value

If successful, the exec... functions do not return. On error, the exec... functions return -1, and the global variable errno is set to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found
ENOEXEC	Exec format error
ENOMEM	Not enough memory

Example

```
/* execl() example */
#include <stdio.h>
#include <process.h>

int main(int argc, char *argv[])
{
```

```
  int loop;
  printf("%s running...\n\n", argv[0]);

  if (argc == 1) {    /* check for only one command-line parameter */
    printf("%s calling itself again...\n", argv[0]);
    execl(argv[0], argv[0], "ONE", "TWO", "THREE", NULL);
    perror("EXEC:");
    exit(1);
  }
  printf("%s called with arguments:\n", argv[0]);
  for (loop = 1; loop <= argc; loop++)
    puts(argv[loop]);       /* Display all command-line parameters */
  return 0;
}
```

3.1.4.22.14 getpid

Header File

process.h

Category

Process Control Routines

Prototype

```
unsigned getpid(void)
```

Description

Gets the process ID of a program.

This function returns the current process ID--an integer that uniquely identifies the process.

Return Value

getpid returns the current process' ID.

Example

```
#include <stdio.h>
#include <process.h>
int main()
{
   printf("This program's process identification number (PID) "
          "number is %X\n", getpid());
   printf("Note: under DOS it is the PSP segment\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.22.15 spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe, _wspawnl, _wspawnle, _wspawnlp, _wspawnlpe, _wspawnv, _wspawnve, _wspawnvp, _wspawnvpe

Header File

process.h

Category

Process Control Routines

Prototype

```
int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
int _wspawnl(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL);
int spawnle(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char *envp[]);
int _wspawnle(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL, wchar_t *envp[]);
int spawnlp(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
int _wspawnlp(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL);
int spawnlpe(int mode, char *path, char *arg0, arg1, ..., argn, NULL, char *envp[]);
int _wspawnlpe(int mode, wchar_t *path, wchar_t *arg0, arg1, ..., argn, NULL, wchar_t *envp[]);
int spawnv(int mode, char *path, char *argv[]);
int _wspawnv(int mode, wchar_t *path, wchar_t *argv[]);
int spawnve(int mode, char *path, char *argv[], char *envp[]);
int _wspawnve(int mode, wchar_t *path, wchar_t *argv[], wchar_t *envp[]);
int spawnvp(int mode, char *path, char *argv[]);
int _wspawnvp(int mode, wchar_t *path, wchar_t *argv[]);
int spawnvpe(int mode, char *path, char *argv[], char *envp[]);
int _wspawnvpe(int mode, wchar_t *path, wchar_t *argv[], wchar_t *envp[]);
```

Note: In spawnle, spawnlpe, spawnv, spawnve, spawnvp, and spawnvpe, the last string must be NULL.

Description

The functions in the spawn... family create and run (execute) other files, known as child processes. There must be sufficient memory available for loading and executing a child process.

The value of mode determines what action the calling function (the parent process) takes after the spawn... call. The possible values of mode are

P_WAIT	Puts parent process on hold until child process completes execution.
P_NOWAIT	Continues to run parent process while child process runs. The child process ID is returned, so that the parent can wait for completion using cwait or wait.
P_NOWAITO	Identical to P_NOWAIT except that the child process ID isn't saved by the operating system, so the parent process can't wait for it using cwait or wait.
P_DETACH	Identical to P_NOWAITO, except that the child process is executed in the background with no access to the keyboard or the display.
P_OVERLAY	Overlays child process in memory location formerly occupied by parent. Same as an exec... call.

path is the file name of the called child process. The spawn... function calls search for path using the standard operating system search algorithm:

- If there is no extension or no period, they search for an exact file name. If the file is not found, they search for files first with the extension EXE, then COM, and finally BAT.

- If an extension is given, they search only for the exact file name.
- If only a period is given, they search only for the file name with no extension.
- If path does not contain an explicit directory, spawn... functions that have the **p** suffix search the current directory, then the directories set with the operating system PATH environment variable.

The suffixes p, l, and v, and e added to the spawn... "family name" specify that the named function operates with certain capabilities.

p	**The function searches for the file in those directories specified by the PATH environment variable. Without the p suffix, the function searches only the current working directory.**
l	The argument pointers arg0, arg1, ..., argn are passed as separate arguments. Typically, the l suffix is used when you know in advance the number of arguments to be passed.
v	The argument pointers argv[0], ..., arg[n] are passed as an array of pointers. Typically, the v suffix is used when a variable number of arguments is to be passed.
e	The argument envp can be passed to the child process, letting you alter the environment for the child process. Without the e suffix, child processes inherit the environment of the parent process.

Each function in the spawn... family must have one of the two argument-specifying suffixes (either l or v). The path search and environment inheritance suffixes (p and e) are optional.

For example:

- spawnl takes separate arguments, searches only the current directory for the child, and passes on the parent's environment to the child.
- spawnvpe takes an array of argument pointers, incorporates PATH in its search for the child process, and accepts the envp argument for altering the child's environment.

The spawn... functions must pass at least one argument to the child process (arg0 or argv[0]). This argument is, by convention, a copy of path. (Using a different value for this 0 argument won't produce an error.) If you want to pass an empty argument list to the child process, then arg0 or argv[0] must be NULL.

When the l suffix is used, arg0 usually points to path, and arg1, ..., argn point to character strings that form the new list of arguments. A mandatory null following argn marks the end of the list.

When the e suffix is used, you pass a list of new environment settings through the argument envp. This environment argument is an array of character pointers. Each element points to a null-terminated character string of the form

```
envvar = value
```

where envvar is the name of an environment variable, and value is the string value to which envvar is set. The last element in envp[] is null. When envp is null, the child inherits the parents' environment settings.

The combined length of arg0 + arg1 + ... + argn (or of argv[0] + argv[1] + ... + argv[n]), including space characters that separate the arguments, must be less than 260 bytes for Windows (128 for DOS). Null-terminators are not counted.

When a spawn... function call is made, any open files remain open in the child process.

Return Value

When successful, the spawn... functions, where mode is P_WAIT, return the child process' exit status (0 for a normal termination). If the child specifically calls exit with a nonzero argument, its exit status can be set to a nonzero value.

If mode is P_NOWAIT or P_NOWAITO, the spawn functions return the process ID of the child process. The ID obtained when using P_NOWAIT can be passed to cwait.

On error, the spawn... functions return -1, and the global variable errno is set to one of the following values:

E2BIG	Arg list too long
EINVAL	Invalid argument
ENOENT	Path or file name not found
ENOEXEC	Exec format error

ENOMEM	Not enough memory

Example

```
#include <process.h>
#include <stdio.h>
void spawnl_example(void)
{
   int result;
   result = spawnl(P_WAIT, "bcc32.exe", "bcc32.exe", NULL);
   if (result == -1)
   {
      perror("Error from spawnl");
      exit(1);
   }
}
int main(void)
{
  spawnl_example();
  return 0;
}
```

3.1.4.22.16 wait

Header File

process.h

Category

Process Control Routines

Prototype

```
int wait(int *statloc);
```

Description

Waits for one or more child processes to terminate.

The wait function waits for one or more child processes to terminate. The child processes must be those created by the calling program; wait cannot wait for grandchildren (processes spawned by child processes). If statloc is not NULL, it points to location where wait will store the termination status.

If the child process terminated normally (by calling exit, or returning from main), the termination status word is defined as follows:

Bits 0-7	**Zero.**
Bits 8-15	The least significant byte of the return code from the child process. This is the value that is passed to exit, or is returned from main. If the child process simply exited from main without returning a value, this value will be unpredictable. If the child process terminated abnormally, the termination status word is defined as follows:
Bits 0-7	Termination information about the child:

Bits 8-15	**Zero.**

Return Value

When wait returns after a normal child process termination it returns the process ID of the child.

When wait returns after an abnormal child termination it returns -1 to the parent and sets errno to EINTR.

If wait returns without a child process completion it returns a -1 value and sets errno to:

ECHILD	No child process exists

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.23 setjmp.h

The following functions, macros, and classes are provided in `setjmp.h`:

Topics

Name	Description
longjmp (see page 994)	**Header File** setjmp.h **Category** Miscellaneous Routines **Prototype** `void longjmp(jmp_buf jmpb, int retval);` **Description** Performs nonlocal goto. A call to longjmp restores the task state captured by the last call to setjmp with the argument jmpb. It then returns in such a way that setjmp appears to have returned with the value retval. A Win32 task state includes: • Register variables • EBX, EDI, ESI • Stack pointer (ESP) • Frame pointer (EBP) • No segment registers are saved • Flags are not saved A task state is complete enough that setjmp and longjmp can be used to implement co-routines. setjmp must be called before longjmp. The... more (see page 994)

setjmp (see page 995)	**Header File** setjmp.h **Category** Miscellaneous Routines **Prototype** `int setjmp(jmp_buf jmpb);` **Description** Sets up for nonlocal goto. setjmp captures the complete task state in jmpb and returns 0. A later call to longjmp with jmpb restores the captured task state and returns in such a way that setjmp appears to have returned with the value val. Under Win32, a task state includes: • No segment registers are saved • Register variables • EBX, EDI, ESI • Stack pointer (ESP) • Frame pointer (EBP) • Flags are not saved A task state is complete enough that setjmp can be used to implement co-routines. setjmp must be called before... more (see page 995)

3.1.4.23.1 longjmp

Header File

setjmp.h

Category

Miscellaneous Routines

Prototype

```
void longjmp(jmp_buf jmpb, int retval);
```

Description

Performs nonlocal goto.

A call to longjmp restores the task state captured by the last call to setjmp with the argument jmpb. It then returns in such a way that setjmp appears to have returned with the value retval.

A Win32 task state includes:

- Register variables
- EBX, EDI, ESI
- Stack pointer (ESP)
- Frame pointer (EBP)
- No segment registers are saved
- Flags are not saved

A task state is complete enough that setjmp and longjmp can be used to implement co-routines.

setjmp must be called before longjmp. The routine that called setjmp and set up jmpb must still be active and cannot have returned before the longjmp is called. If this happens, the results are unpredictable.

longjmp cannot pass the value 0; if 0 is passed in retval, longjmp will substitute 1.

Return Value

None.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>
void subroutine(jmp_buf);
int main(void)
{

    int value;
    jmp_buf jumper;
    value = setjmp(jumper);
    if (value != 0)
    {
       printf("Longjmp with value %d\n", value);
       exit(value);
    }
    printf("About to call subroutine ... \n");
    subroutine(jumper);
    return 0;
}
void subroutine(jmp_buf jumper)
{
    longjmp(jumper,1);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.23.2 setjmp

Header File

setjmp.h

Category

Miscellaneous Routines

Prototype

```
int setjmp(jmp_buf jmpb);
```

Description

Sets up for nonlocal goto.

setjmp captures the complete task state in jmpb and returns 0.

A later call to longjmp with jmpb restores the captured task state and returns in such a way that setjmp appears to have returned with the value val.

Under Win32, a task state includes:

- No segment registers are saved
- Register variables

- EBX, EDI, ESI
- Stack pointer (ESP)
- Frame pointer (EBP)
- Flags are not saved

A task state is complete enough that setjmp can be used to implement co-routines.

setjmp must be called before longjmp. The routine that calls setjmp and sets up jmpb must still be active and cannot have returned before the longjmp is called. If it has returned, the results are unpredictable.

setjmp is useful for dealing with errors and exceptions encountered in a low-level subroutine of a program.

Return Value

setjmp returns 0 when it is initially called. If the return is from a call to longjmp, setjmp returns a nonzero value (as in the example).

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>
void subroutine(jmp_buf);
int main(void)
{

    int value;
    jmp_buf jumper;
    value = setjmp(jumper);
    if (value != 0)
    {
       printf("Longjmp with value %d\n", value);
       exit(value);
    }
    printf("About to call subroutine ... \n");
    subroutine(jumper);
    return 0;
}
void subroutine(jmp_buf jumper)
{
    longjmp(jumper,1);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.24 share.h

The following functions, macros, and classes are provided in `share.h`:

Topics

Name	Description
SH_xxxx #defines (see page 997)	**Header File** share.h **Description** File-sharing mode for use with sopen.

3.1.4.24.1 SH_xxxx #defines

Header File

share.h

Description

File-sharing mode for use with sopen.

SH_COMPAT	Sets compatibility mode:
	Allows other opens with SH_COMPAT. The call will fail if the file has already been opened in any other shared mode.
SH_DENYNONE	Permits read/write access
	Allows other shared opens to the file, but not other SH_COMPAT opens
SH_DENYNO	Permits read/write access (provided for compatibility)
SH_DENYRD	Denies read access. Allows only writes from any other open to the file
SH_DENYRW	Denies read/write access. Only the current handle may have access to the file
SH_DENYWR	Denies write access. Allows only reads from any other open to the file
O_NOINHERIT	The file is not passed to child programs

These file-sharing attributes are in addition to any locking performed on the files.

3.1.4.25 signal.h

The following functions, macros, and classes are provided in `signal.h`:

Topics

Name	Description
SIGxxxx #defines (see page 998)	**Header File** signal.h **Description** Signal types used by raise and signal.
SIG_xxx #defines (see page 998)	**Header File** signal.h **Description** Predefined functions for handling signals generated by raise or by external events.
raise (see page 999)	**Header File** signal.h **Category** Process Control Routines **Prototype** `int raise(int sig);` **Description** Sends a software signal to the executing program. raise sends a signal of type sig to the program. If the program has installed a signal handler for the signal type specified by sig, that handler will be executed. If no handler has been installed, the default action for that signal type will be taken. The signal types currently defined in signal.h are noted here:

signal (C RTL) (see page 1000)	**Header File** signal.h **Category** Process Control Routines **Prototype** `void (_USERENTRY *signal(int sig, void (_USERENTRY *func)(int sig[, int subcode])))(int);` **Description** Specifies signal-handling actions. signal determines how receipt of signal number sig will subsequently be treated. You can install a user-specified handler routine (specified by the argument func) or use one of the two predefined handlers, SIG_DFL and SIG_IGN, in signal.h. The function func must be declared with the _USERENTRY calling convention. A routine that catches a signal (such as a floating point) also clears the signal. To continue to receive signals, a signal handler must be reinstalled by calling signal again.... more (see page 1000)

3.1.4.25.1 SIGxxxx #defines

Header File

signal.h

Description

Signal types used by raise and signal.

Signal	Note	Meaning	Default Action
SIGABRT	(*)	Abnormal termination	= to calling _exit(3)
SIGFPE		Bad floating-point operation	= to calling _exit(1)
		Arithmetic error caused by	
		division by 0, invalid operation, etc.	
SIGILL		Illegal operation	= to calling _exit(1)
SIGINT		Control-C interrupt	= to calling _exit(3)
SIGSEGV		Invalid access to storage	= to calling _exit(1)
SIGTERM	(*)	Request for program termination	= to calling _exit(1)

(*) Signal types marked with a (*) aren't generated by Borland C++ during normal operation. However, they can be generated with raise.

3.1.4.25.2 SIG_xxx #defines

Header File

signal.h

Description

Predefined functions for handling signals generated by raise or by external events.

Name	Meaning
SIG_DFL	Terminate the program
SIG_IGN	No action, ignore signal
SIG_ERR	Return error code

3.1.4.25.3 raise

Header File

signal.h

Category

Process Control Routines

Prototype

```
int raise(int sig);
```

Description

Sends a software signal to the executing program.

raise sends a signal of type sig to the program. If the program has installed a signal handler for the signal type specified by sig, that handler will be executed. If no handler has been installed, the default action for that signal type will be taken.

The signal types currently defined in signal.h are noted here:

SIGABRT	Abnormal termination
SIGFPE	Bad floating-point operation
SIGILL	Illegal instruction
SIGINT	Ctrl-C interrupt
SIGSEGV	Invalid access to storage
SIGTERM	Request for program termination
SIGUSR1	User-defined signal
SIGUSR2	User-defined signal
SIGUSR3	User-defined signal
SIGBREAK	Ctrl-Break interrupt

Note: SIGABRT isn't generated by C++Builder during normal operation. It can, however, be generated by abort, raise, or unhandled exceptions.

Return Value

On success, raise returns 0.

On error it returns nonzero.

Example

```
#include <signal.h>
int main(void)
{
    int a, b;
    a = 10;
    b = 0;
    if (b == 0)
    /* preempt divide by zero error */
       raise(SIGFPE);
    a = a / b;
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.25.4 signal (C RTL)

Header File

signal.h

Category

Process Control Routines

Prototype

```
void (_USERENTRY *signal(int sig, void (_USERENTRY *func)(int sig[, int subcode])))(int);
```

Description

Specifies signal-handling actions.

signal determines how receipt of signal number sig will subsequently be treated. You can install a user-specified handler routine (specified by the argument func) or use one of the two predefined handlers, SIG_DFL and SIG_IGN, in signal.h. The function func must be declared with the _USERENTRY calling convention.

A routine that catches a signal (such as a floating point) also clears the signal. To continue to receive signals, a signal handler must be reinstalled by calling signal again.

SIG_DFL	Terminates the program
SIG_ERR	Indicates an error return from signal
SIG_IGN	Ignore this type signal

The following table shows signal types and their defaults:

SIGBREAK	Keyboard must be in raw mode.
SIGABRT	Abnormal termination. Default action is equivalent to calling _exit(3).
SIGFPE	Arithmetic error caused by division by 0, invalid operation, and the like. Default action is equivalent to calling _exit(1).
SIGILL	Illegal operation. Default action is equivalent to calling _exit(1).
SIGINT	Ctrl-C interrupt. Default action is equivalent to calling _exit(3).
SIGSEGV	Illegal storage access. Default action is equivalent to calling _exit(1).
SIGTERM	Request for program termination. Default action is equivalent to calling _exit(1).
	User-defined signals can be generated only by calling raise. Default action is to ignore the signal.

signal.h defines a type called sig_atomic_t, the largest integer type the processor can load or store atomically in the presence of asynchronous interrupts (this is a 32-bit integer -- a Borland C++ integer).

When a signal is generated by the raise function or by an external event, the following two things happen:

- If a user-specified handler has been installed for the signal, the action for that signal type is set to SIG_DFL.

- The user-specified function is called with the signal type as the parameter.

User-specified handler functions can terminate by a return or by a call to abort, _exit, exit, or longjmp. If your handler function is expected to continue to receive and handle more signals, you must have the handler function call signal again.

Borland C++ implements an extension to ANSI C when the signal type is SIGFPE, SIGSEGV, or SIGILL. The user-specified handler function is called with one or two extra parameters. If SIGFPE, SIGSEGV, or SIGILL has been raised as the result of an explicit call to the raise function, the user-specified handler is called with one extra parameter, an integer specifying that the handler is being explicitly invoked. The explicit activation values for SIGFPE, SIGSEGV and SIGILL are as follows

Note: Declarations of these types are defined in float.h.

SIGFPE	FPE_EXPLICITGEN
SIGSEGV	SEGV_EXPLICITGEN
SIGILL	ILL_EXPLICITGEN

If SIGFPE is raised because of a floating-point exception, the user handler is called with one extra parameter that specifies the FPE_xxx type of the signal. If SIGSEGV, SIGILL, or the integer-related variants of SIGFPE signals (FPE_INTOVFLOW or FPE_INTDIV0) are raised as the result of a processor exception, the user handler is called with two extra parameters:

1.The SIGFPE, SIGSEGV, or SIGILL exception type (see float.h for all these types). This first parameter is the usual ANSI signal type.

2.An integer pointer into the stack of the interrupt handler that called the user-specified handler. This pointer points to a list of the processor registers saved when the exception occurred. The registers are in the same order as the parameters to an interrupt function; that is, EBP, EDI, ESI, EDS, ES, EDX, ECX, EBX, EAX, EIP, CS, EFL. To have a register value changed when the handler returns, change one of the locations in this list.

For example, to have a new SI value on return, do something like this:

```
*((int*)list_pointer + 2) = new_SI_value;
```

In this way, the handler can examine and make any adjustments to the registers that you want.

The following SIGFPE-type signals can occur (or be generated). They correspond to the exceptions that the 80x87 family is capable of detecting, as well as the "INTEGER DIVIDE BY ZERO" and the "INTERRUPT ON OVERFLOW" on the main CPU. (The declarations for these are in float.h.)

FPE_INTOVFLOW	INTO executed with OF flag set
FPE_INTDIV0	Integer divide by zero
FPE_INVALID	Invalid operation
FPE_ZERODIVIDE	Division by zero
FPE_OVERFLOW	Numeric overflow
FPE_UNDERFLOW	Numeric underflow
FPE_INEXACT	Precision
FPE_EXPLICITGEN	User program executed raise(SIGFPE)
FPE_STACKFAULT	Floating-point stack overflow or underflow
FPE_STACKFAULT	Stack overflow

The FPE_INTOVFLOW and FPE_INTDIV0 signals are generated by integer operations, and the others are generated by floating-point operations. Whether the floating-point exceptions are generated depends on the coprocessor control word, which can be modified with _control87. Denormal exceptions are handled by Borland C++ and not passed to a signal handler.

The following SIGSEGV-type signals can occur:

SEGV_BOUND	Bound constraint exception
SEGV_EXPLICITGEN	raise(SIGSEGV) was executed

The following SIGILL-type signals can occur:

ILL_EXECUTION	Illegal operation attempted
ILL_EXPLICITGEN	raise(SIGILL) was executed

When the signal type is SIGFPE, SIGSEGV, or SIGILL, a return from a signal handler is generally not advisable if the state of the floating point processor is corrupt, the results of an integer division are wrong, an operation that shouldn't have overflowed did, a bound instruction failed, or an illegal operation was attempted. The only time a return is reasonable is when the handler alters the registers so that a reasonable return context exists or the signal type indicates that the signal was generated explicitly (for example, FPE_EXPLICITGEN, SEGV_EXPLICITGEN, or ILL_EXPLICITGEN). Generally in this case you would print an error message and terminate the program using _exit, exit, or abort. If a return is executed under any other conditions, the program's action will probably be unpredictable.

Note: Take special care when using the signal function in a multithread program. The SIGINT, SIGTERM, and SIGBREAK signals can be used only by the main thread (thread one) in a non-Win32 application. When one of these signals occurs, the currently executing thread is suspended, and control transfers to the signal handler (if any) set up by thread one. Other signals can be handled by any thread.

Note: A signal handler should not use C++ runtime library functions, because a semaphore deadlock might occur. Instead, the handler should simply set a flag or post a semaphore, and return immediately.

Return Value

On success, signal returns a pointer to the previous handler routine for the specified signal type.

On error, signal returns SIG_ERR, and the external variable errno is set to EINVAL.

Example

```
/* signal example */
/*
    This example installs a signal handler routine for SIGFPE,
    catches an integer overflow condition, makes an adjustment to AX
    register, and returns.  This example program MAY cause your computer
    to crash, and will produce runtime errors depending on which memory
    model is used.
*/
#pragma inline
#include <stdio.h>
#include <signal.h>
#ifdef __cplusplus
   typedef void (*fptr)(int);
#else
   typedef void (*fptr)();
#endif
void Catcher(int *reglist)
{
   signal(SIGFPE, (fptr)Catcher);  //  ******reinstall signal handler
   printf("Caught it!\n"); *(reglist + 8) = 3; /* make return AX = 3 */
}
int main(void)
{
   signal(SIGFPE, (fptr)Catcher);  /* cast Catcher to appropriate type */
   asm     mov     ax,07FFFH       /* AX = 32767 */
   asm     inc     ax              /* cause overflow */
   asm     into                    /* activate handler */
   /* The handler set AX to 3 on return. If that had not happened,
```

```
      there would have been another exception when the next 'into'
      executed after the 'dec' instruction. */
   asm     dec     ax              /* no overflow now */
   asm     into                    /* doesn't activate */
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.26 stdarg.h

The following functions, macros, and classes are provided in `stdarg.h`:

Topics

Name	Description

va_arg, va_end, va_start (see page 1003)	**Header File**
	stdarg.h
	Category
	Variable Argument List Routines
	Prototype
	`void va_start(va_list ap, lastfix);`
	`type va_arg(va_list ap, type);`
	`void va_end(va_list ap);`
	Description
	Implement a variable argument list.
	Some C functions, such as vfprintf and vprintf, take variable argument lists in addition to taking a number of fixed (known) parameters. The va_arg, va_end, and va_start macros provide a portable way to access these argument lists. They are used for stepping through a list of arguments when the called function does not know the number and types of the arguments being passed.
	The header file stdarg.h declares one type (**va_list**) and... more (see page 1003)

3.1.4.26.1 va_arg, va_end, va_start

Header File

stdarg.h

Category

Variable Argument List Routines

Prototype

```
void va_start(va_list ap, lastfix);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Description

Implement a variable argument list.

Some C functions, such as vfprintf and vprintf, take variable argument lists in addition to taking a number of fixed (known) parameters. The va_arg, va_end, and va_start macros provide a portable way to access these argument lists. They are used for stepping through a list of arguments when the called function does not know the number and types of the arguments being

passed.

The header file stdarg.h declares one type (**va_list**) and three macros (va_start, va_arg, and va_end).

- **va_list**: This array holds information needed by va_arg and va_end. When a called function takes a variable argument list, it declares a variable ap of type **va_list**.
- **va_start**: This routine (implemented as a macro) sets ap to point to the first of the variable arguments being passed to the function. va_start must be used before the first call to va_arg or va_end.
- va_start takes two parameters: ap and lastfix. (ap is explained under va_list in the preceding paragraph; lastfix is the name of the last fixed parameter being passed to the called function.)
- **va_arg:** This routine (also implemented as a macro) expands to an expression that has the same type and value as the next argument being passed (one of the variable arguments). The variable ap to va_arg should be the same ap that va_start initialized.

  Note: Because of default promotions, you cannot use char, unsigned char

  , or **float** types with va_arg.

  Note: The first time va_arg is used, it returns the first argument in the list. Each successive time va_arg is used, it returns the next argument in the list. It does this by first dereferencing ap, and then incrementing ap to point to the following item. va_arg uses the type to both perform the dereference and to locate the following item. Each successive time va_arg is invoked, it modifies ap to point to the next argument in the list.
- **va_end:** This macro helps the called function perform a normal return. va_end might modify ap in such a way that it cannot be used unless va_start is recalled. va_end should be called after va_arg has read all the arguments; failure to do so might cause strange, undefined behavior in your program.

Return Value

va_start and va_end return no values; va_arg returns the current argument in the list (the one that ap is pointing to).

Example

```
#include <stdio.h>
#include <stdarg.h>
/* calculate sum of a 0 terminated list */
void sum(char *msg, ...)
{
   int total = 0;
   va_list ap;
   int arg;
   va_start(ap, msg);
   while ((arg = va_arg(ap,int)) != 0) {
      total += arg;
   }
   printf(msg, total);
   va_end(ap);
}
int main(void) {
   sum("The total of 1+2+3+4 is %d\n", 1,2,3,4,0);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.27 stddef.h

The following functions, macros, and classes are provided in `stddef.h`:

Topics

Name	Description
NULL #define (see page 1005)	**Header File** stddef.h **Description** Null pointer constant that is compatible with any data object pointer. It is not compatible with function pointers. When a pointer is equivalent to NULL it is guaranteed not to point to any data object defined within the program.
_threadid (see page 1005)	**Header File** stddef.h **Syntax** `extern long _threadid;` **Description** _threadid is a long integer that contains the ID of the currently executing thread. It is implemented as a macro, and should be declared only by including stddef.h.
offsetof (see page 1006)	**Header File** stddef.h **Category** Memory Routines **Prototype** `size_t offsetof(struct_type, struct_member);` **Description** Gets the byte offset to a structure member. offsetof is available only as a macro. The argument struct_type is a **struct** type. struct_member is any element of the **struct** that can be accessed through the member selection operators or pointers. If struct_member is a bit field, the result is undefined. See also sizeof for more information on memory allocation and alignment of structures. **Return Value** offsetof returns the number of bytes from the start of the structure to the start of the named structure member. **Portability**

3.1.4.27.1 NULL #define

Header File

stddef.h

Description

Null pointer constant that is compatible with any data object pointer. It is not compatible with function pointers. When a pointer is equivalent to NULL it is guaranteed not to point to any data object defined within the program.

3.1.4.27.2 _threadid

Header File

stddef.h

Syntax

```
extern long _threadid;
```

Description

_threadid is a long integer that contains the ID of the currently executing thread. It is implemented as a macro, and should be declared only by including stddef.h.

3.1.4.27.3 offsetof

Header File

stddef.h

Category

Memory Routines

Prototype

```
size_t offsetof(struct_type, struct_member);
```

Description

Gets the byte offset to a structure member.

offsetof is available only as a macro. The argument struct_type is a **struct** type. struct_member is any element of the **struct** that can be accessed through the member selection operators or pointers.

If struct_member is a bit field, the result is undefined.

See also sizeof for more information on memory allocation and alignment of structures.

Return Value

offsetof returns the number of bytes from the start of the structure to the start of the named structure member.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+	+	+

3.1.4.28 stdio.h

The following functions, macros, and classes are provided in `stdio.h`:

Topics

Name	Description
BUFSIZ #define (see page 1023)	**Header File** stdio.h **Description** Default buffer size used by setbuf function.
_F_xxxx #defines (see page 1023)	**Header File** stdio.h **Description** File status flags of streams
OPEN_MAX #define (see page 1024)	**Header File** stdio.h **Description** Number of files that can be open simultaneously.
L_ctermid #define (see page 1024)	**Header File** stdio.h **Description** The length of a device id string.

L_tmpnam #define (see page 1024)	**Header File** stdio.h **Description** Size of an array large enough to hold a temporary file name string.
TMP_MAX #define (see page 1024)	**Header File** stdio.h **Description** Maximum number of unique file names.
_IOxxx #defines (see page 1025)	**Header File** stdio.h **Description** Constants for defining buffering style to be used with a file.

_fsopen, _wfsopen (see page 1025)	**Header File**
	stdio.h, share.h
	Category
	Input/output Routines
	Prototype
	`FILE *_fsopen(const char *filename, const char *mode, int shflag);`
	`FILE *_wfsopen(const wchar_t *filename, const wchar_t *mode, int shflag);`
	Description
	Opens a stream with file sharing.
	_fsopen opens the file named by filename and associates a stream with it. _fsopen returns a pointer that is used to identify the stream in subsequent operations.
	The mode string used in calls to _fsopen is one of the following values:

_pclose (see page 1027)	**Header File** stdio.h **Category** Input/output Routines, Process Control Routines **Prototype** `int _pclose(FILE * stream);` **Description** Waits for piped command to complete. _pclose closes a pipe stream created by a previous call to _popen, and then waits for the associated child command to complete. **Return Value** On success, _pclose returns the termination status of the child command. This is the same value as the termination status returned by cwait, except that the high and low order bytes of the low word are swapped. On error, it returns -1. **Portability**

_popen, _wpopen (see page 1027)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`FILE *_popen (const char *command, const char *mode);`
	`FILE *_wpopen (const wchar_t *command, const wchar_t *mode);`
	Description
	Creates a command processor pipe.
	The _popen function creates a pipe to the command processor. The command processor is executed asynchronously, and is passed the command line in command. The mode string specifies whether the pipe is connected to the command processor's standard input or output, and whether the pipe is to be opened in binary or text mode.
	The mode string can take one of the following values:

_snprintf;_snwprintf (see page 1028)	**Header File**
	stdio.h
	Category
	Memory and String Manipulation Routines
	Syntax
	`int _snprintf(char* buffer, size_t nsize, const char* format, ...);`
	`int _snwprintf(wchar_t* buffer, size_t nsize, const wchar_t* format, ...);`
	Description
	Sends formatted output to a string of a maximum length specified by nsize. _snprintf and _snwprintf are Microsoft compatible with the _snprintf and _snprintfw functions, respectively.
	If the number of bytes to output is:
	• < nsize, then all of the characters have been written, including the terminating '\0' character.
	• == nsize, then nsize characters are written with no terminating '\0' character.
	If nsize is 0, then the string will not be... more (see page 1028)

_vsnprintf;_vsnwprintf (see page 1029)	**Header File**
	stdio.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int _vsnprintf(char* buffer, size_t nsize, const char* format, va_list param);`
	`int _vsnwprintf(wchar_t* buffer, size_t nsize, const wchar_t* format, va_list param);`
	Description
	Sends formatted output to a string of a maximum length specified by nsize. _vsnprintf and _vsnwprintf are Microsoft compatible with the _vsnprintf and _vsnprintfw functions, respectively.
	If the number of bytes to output is:
	• < nsize, then all of the characters have been written, including the terminating '\0' character.
	• == nsize, then nsize characters are written with no terminating '\0' character.
	If nsize is 0, then the string will... more (see page 1029)

clearerr (see page 1029)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `void clearerr(FILE *stream);` **Description** Resets error indication. clearerr resets the named stream's error and end-of-file indicators to 0. Once the error indicator is set, stream operations continue to return error status until a call is made to clearerr or rewind. The end-of-file indicator is reset with each input operation. **Return Value** None. **Example**
EOF #define (see page 1030)	**Header File** stdio.h **Description** A constant indicating that end-of-file has been reached on a file.

fclose (see page 1030)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int fclose(FILE *stream);` **Description** Closes a stream. fclose closes the named stream. All buffers associated with the stream are flushed before closing. System-allocated buffers are freed upon closing. Buffers assigned with setbuf or setvbuf are not automatically freed. (But if setvbuf is passed null for the buffer pointer it will free it upon close.) **Return Value** fclose returns 0 on success. It returns EOF if any errors were detected. **Portability**
_fcloseall (see page 1031)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int _fcloseall(void);` **Description** Closes open streams. _fcloseall closes all open streams except stdauxstdstreams When _fcloseall flushes the associated buffers before closing a stream. The buffers allocated by the system are released. **Note:** stdprn and stdaux streams are not available in Win32. **Return Value** _fcloseall returns the total number of streams it closed. The _fcloseall function returns EOF if any errors were detected. **Example**

_fdopen, _wfdopen (see page 1032)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`FILE *_fdopen(int handle, char *type);`
	`FILE *_wfdopen(int handle, wchar_t *type);`
	Description
	Associates a stream with a file handle.
	_fdopen associates a stream with a file handle obtained from creat, dup, dup2, or open.
	The type of stream must match the mode of the open handle.
	The type string used in a call to _fdopen is one of the following values:

feof (see page 1033)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int feof(FILE *stream);` **Description** Detects end-of-file on a stream. feof is a macro that tests the given stream for an end-of-file indicator. Once the indicator is set read operations on the file return the indicator until rewind is called or the file is closed. The end-of-file indicator is reset with each input operation. **Return Value** feof returns nonzero if an end-of-file indicator was detected on the last input operation on the named stream and 0 if end-of-file has not been reached. **Example**

ferror (see page 1034)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int ferror(FILE *stream);` **Description** Detects errors on stream. ferror is a macro that tests the given stream for a read or write error. If the stream's error indicator has been set it remains set until clearerr or rewind is called or until the stream is closed. **Return Value** ferror returns nonzero if an error was detected on the named stream. **Example**
fflush (see page 1035)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int fflush(FILE *stream);` **Description** Flushes a stream. If the given stream has buffered output fflush writes the output for stream to the associated file. The stream remains open after fflush has executed. fflush has no effect on an unbuffered stream. **Return Value** fflush returns 0 on success. It returns EOF if any errors were detected. **Example**

fgetc, fgetwc (see page 1036)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int fgetc(FILE *stream);`
	`wint_t fgetwc(FILE *stream);`
	Description
	Gets character from stream.
	fgetc returns the next character on the named input stream.
	Return Value
	On success fgetc returns the character read after converting it to an **int** without sign extension. On end-of-file or error it returns EOF.
	Example

_fgetchar, _fgetwchar (see page 1037)	**Header File**
	stdio.h
	Category
	Console I/O Routines
	Prototype
	`int _fgetchar(void);`
	`wint_t _fgetwchar(void);`
	Description
	Reads a character from stdin.
	_fgetchar returns the next character from stdin. It is defined as fgetc(stdin).
	Note: For Win32 GUI applications, stdin must be redirected.
	Return Value
	On success _fgetchar returns the character read after converting it to an **int** without sign extension. On end-of-file or error it returns EOF.
	Example

fgetpos (see page 1037)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int fgetpos(FILE *stream, fpos_t *pos);` **Description** Gets the current file pointer. fgetpos stores the position of the file pointer associated with the given stream in the location pointed to by pos. The exact value is unimportant; its value is opaque except as a parameter to subsequent fsetpos calls. **Return Value** On success fgetpos returns 0. On failure it returns a nonzero value and sets the global variable errno to

fgets, fgetws (see page 1039)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`char *fgets(char *s, int n, FILE *stream);`
	`wchar_t *fgetws(wchar_t *s, int n, FILE *stream); // Unicode version`
	Description
	Gets a string from a stream.
	fgets reads characters from stream into the string s. The function stops reading when it reads either n - 1 characters or a newline character whichever comes first. fgets retains the newline character at the end of s. A null byte is appended to s to mark the end of the string.
	Return Value
	On success fgets returns the string pointed to by s; it returns NULL on end-of-file... more (see page 1039)

_fileno (see page 1040)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int _fileno(FILE *stream);` **Description** Gets the file handle. _fileno is a macro that returns the file handle for the given stream. If stream has more than one handle _fileno returns the handle assigned to the stream when it was first opened. **Return Value** _fileno returns the integer file handle associated with stream. **Example**
_flushall (see page 1040)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int _flushall(void);` **Description** Flushes all streams. _flushall clears all buffers associated with open input streams and writes all buffers associated with open output streams to their respective files. Any read operation following _flushall reads new data into the buffers from the input files. Streams stay open after _flushall executes. **Return Value** _flushall returns an integer the number of open input and output streams. **Example**

fopen, _wfopen (see page 1041)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`FILE *fopen(const char *filename, const char *mode);`
	`FILE *_wfopen(const wchar_t *filename, const wchar_t *mode);`
	Description
	Opens a stream.
	fopen opens the file named by filename and associates a stream with it. fopen returns a pointer to be used to identify the stream in subsequent operations.
	The mode string used in calls to fopen is one of the following values:

fprintf, fwprintf (see page 1042)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int fprintf(FILE *stream, const char *format[, argument, ...]);`
	`int fwprintf(FILE *stream, const wchar_t *format[, argument, ...]);`
	Description
	Writes formatted output to a stream.
	fprintf accepts a series of arguments applies to each a format specifier contained in the format string pointed to by format and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.
	Note: For details on format specifiers, see printf Format Specifiers.
	Return Value
	fprintf returns the number of bytes output. In the event of error it returns EOF.
	Example

fputc, fputwc (see page 1043)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int fputc(int c, FILE *stream);`
	`wint_t fputwc(wint_t c, FILE *stream);`
	Description
	Puts a character on a stream.
	fputc outputs character c to the named stream.
	Note: For Win32 GUI applications, stdin must be redirected.
	Return Value
	On success, fputc returns the character c. On error, it returns EOF.
	Example

_fputchar, _fputwchar (see page 1044)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int _fputchar(int c);`
	`wint_t _fputwchar(wint_t c);`
	Description
	Outputs a character to stdout.
	_fputchar outputs character c to stdout. _fputchar(c) is the same as fputc(cstdout).
	Note: For Win32 GUI applications, stdout must be redirected.
	Return Value
	On success _fputchar returns the character c.
	On error it returns EOF.
	Example

fputs, fputws (see page 1044)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int fputs(const char *s, FILE *stream);`
	`int fputws(const wchar_t *s, FILE *stream);`
	Description
	Outputs a string on a stream.
	fputs copies the null-terminated string s to the given output stream; it does not append a newline character and the terminating null character is not copied.
	Return Value
	On success fputs returns a non-negative value.
	On error it returns a value of EOF.
	Example

fread (see page 1045)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `size_t fread(void *ptr, size_t size, size_t n, FILE *stream);` **Description** Reads data from a stream. fread reads n items of data each of length size bytes from the given input stream into a block pointed to by ptr. The total number of bytes read is (n * size). **Return Value** On success fread returns the number of items (not bytes) actually read. On end-of-file or error it returns a short count (possibly 0). **Example**

freopen, _wfreopen (see page 1046)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`FILE *freopen(const char *filename, const char *mode, FILE *stream);`
	`FILE *_wfreopen(const wchar_t *filename, const wchar_t *mode, FILE *stream);`
	Description
	Associates a new file with an open stream.
	freopen substitutes the named file in place of the open stream. It closes stream regardless of whether the open succeeds. freopen is useful for changing the file attached to stdin, stdout, or stderr.
	The mode string used in calls to fopen is one of the following values:

fscanf, fwscanf (see page 1047)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int fscanf(FILE *stream, const char *format[, address, ...]);`
	`int fwscanf(FILE *stream, const wchar_t *format[, address, ...]);`
	Description
	Scans and formats input from a stream.
	fscanf scans a series of input fields one character at a time reading from a stream. Then each field is formatted according to a format specifier passed to fscanf in the format string pointed to by format. Finally fscanf stores the formatted input at an address passed to it as an argument following format. The number of format specifiers and addresses must be the same as the number of... more (see page 1047)

fseek (see page 1048)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int fseek(FILE *stream, long offset, int whence);` **Description** Repositions a file pointer on a stream. fseek sets the file pointer associated with stream to a new position that is offset bytes from the file location given by whence. For text mode streams offset should be 0 or a value returned by ftell. whence must be one of the values 0. 1, or 2 which represent three symbolic constants (defined in stdio.h) as follows: fseek discards any character pushed back using ungetc. fseek is used with stream I/O; for file handle I/O use lseek.... more (see page 1048)
fsetpos (see page 1049)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int fsetpos(FILE *stream, const fpos_t *pos);` **Description** Positions the file pointer of a stream. fsetpos sets the file pointer associated with stream to a new position. The new position is the value obtained by a previous call to fgetpos on that stream. It also clears the end-of-file indicator on the file that stream points to and undoes any effects of ungetc on that file. After a call to fsetpos the next operation on the file can be input or output. **Return Value** On success fsetpos returns 0. On failure it returns a nonzero... more (see page 1049)
ftell (see page 1050)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `long int ftell(FILE *stream);` **Description** Returns the current file pointer. ftell returns the current file pointer for stream. The offset is measured in bytes from the beginning of the file (if the file is binary). The value returned by ftell can be used in a subsequent call to fseek. **Return Value** ftell returns the current file pointer position on success. It returns -1L on error and sets the global variable errno to a positive value. In the event of an error return the global variable errno is set to one of the following... more (see page 1050)
fwrite (see page 1051)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);` **Description** Writes to a stream. fwrite appends n items of data each of length size bytes to the given output file. The data written begins at ptr. The total number of bytes written is (n x size). ptr in the declarations is a pointer to any object. **Return Value** On successful completion fwrite returns the number of items (not bytes) actually written. On error it returns a short count. **Example**

getc, getwc (see page 1052)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int getc(FILE *stream);`
	`wint_t getwc(FILE *stream);`
	Description
	Gets character from stream.
	getc returns the next character on the given input stream and increments the stream's file pointer to point to the next character.
	Note: For Win32 GUI applications, stdin must be redirected.
	Return Value
	On success, getc returns the character read, after converting it to an **int** without sign extension.
	On end-of-file or error, it returns EOF.
	Example

getchar, getwchar (see page 1052)	**Header File**
	stdio.h
	Category
	Console I/O Routines
	Prototype
	`int getchar(void);`
	`wint_t getwchar(void);`
	Description
	Gets character from stdin.
	getchar is a macro that returns the next character on the named input stream stdin. It is defined to be getc(stdin).
	Note: Do not use this function in Win32 GUI applications.
	Return Value
	On success, getchar returns the character read, after converting it to an **int** without sign extension.
	On end-of-file or error, it returns EOF.
	Example

gets, _getws (see page 1053)	**Header File**
	stdio.h
	Category
	Console I/O Routines
	Prototype
	`char *gets(char *s);`
	`wchar_t *_getws(wchar_t *s); // Unicode version`
	Description
	Gets a string from stdin.
	gets collects a string of characters terminated by a new line from the standard input stream stdin and puts it into s. The new line is replaced by a null character (\0) in s.
	gets allows input strings to contain certain whitespace characters (spaces, tabs). gets returns when it encounters a new line; everything up to the new line is copied into s.
	The gets function is not length-terminated. If the input string is sufficiently large, data... more (see page 1053)

_getw (see page 1054)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int _getw(FILE *stream);` **Description** Gets an integer from stream. _getw returns the next integer in the named input stream. It assumes no special alignment in the file. _getw should not be used when the stream is opened in text mode. **Return Value** _getw returns the next integer on the input stream. On end-of-file or error, _getw returns EOF. **Note:** Because EOF is a legitimate value for _getw to return, feof or ferror should be used to detect end-of-file or error. **Example**

printf, wprintf (see page 1055)	**Header File**
	stdio.h
	Category
	Console I/O Routines
	Prototype
	`int printf(const char *format[, argument, ...]);`
	`int wprintf(const wchar_t *format[, argument, ...]);`
	Description
	Writes formatted output to stdout.
	The printf function:
	• Accepts a series of arguments
	• Applies to each argument a format specifier contained in the format string *format
	• Outputs the formatted data (to the screen, a stream, stdout, or a string)
	There must be enough arguments for the format. If there are not, the results will be unpredictable and likely disastrous. Excess arguments (more than required by the format) are merely ignored.
	Note: For Win32 GUI applications, stdout must be redirected.... more (see page 1055)

putc, putwc (see page 1057)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int putc(int c, FILE *stream);`
	`wint_t putwc(wint_t c, FILE *stream);`
	Description
	Outputs a character to a stream.
	putc is a macro that outputs the character c to the stream given by stream.
	Return Value
	On success, putc returns the character printed, c.
	On error, putc returns EOF.
	Example

putchar, putwchar (see page 1057)	**Header File**
	stdio.h
	Category
	Console I/O Routines
	Prototype
	`int putchar(int c);`
	`wint_t putwchar(wint_t c);`
	Description
	putchar is a macro defined to be putc(c, stdout).
	Note: For Win32 GUI applications, stdout must be redirected.
	Return Value
	On success, putchar returns the character c. On error, putchar returns EOF.
	Example

puts, _putws (see page 1059)	**Header File**
	stdio.h
	Category
	Console I/O Routines
	Prototype
	`int puts(const char *s);`
	`int _putws(const wchar_t *s);`
	Description
	Outputs a string to stdout.
	puts copies the null-terminated string s to the standard output stream stdout and appends a newline character.
	Note: For Win32 GUI applications, stdout must be redirected.
	Return Value
	On successful completion, puts returns a nonnegative value. Otherwise, it returns a value of EOF.
	Example

_putw (see page 1059)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int _putw(int w, FILE *stream);` **Description** Writes an integer on a stream. _putw outputs the integer w to the given stream. _putw neither expects nor causes special alignment in the file. **Return Value** On success, _putw returns the integer w. On error, _putw returns EOF. Because EOF is a legitimate integer, use ferror to detect errors with _putw. **Example**

remove, _wremove (see page 1060)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int remove(const char *filename);`
	`int _wremove(const wchar_t *filename);`
	Description
	Removes a file.
	remove deletes the file specified by filename. It is a macro that simply translates its call to a call to unlink. If your file is open, be sure to close it before removing it.
	The filename string can include a full path.
	Return Value
	On successful completion, remove returns 0. On error, it returns -1, and the global variable errno is set to one of the following values:

rename, _wrename (see page 1061)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int rename(const char *oldname, const char *newname);`
	`int _wrename(const wchar_t *oldname, const wchar_t *newname);`
	Description
	Renames a file.
	rename changes the name of a file from oldname to newname. If a drive specifier is given in newname, the specifier must be the same as that given in oldname.
	Directories in oldname and newname need not be the same, so rename can be used to move a file from one directory to another. Wildcards are not allowed.
	This function will fail (EEXIST) if either file is currently open in any process.
	Return Value
	On... more (see page 1061)

rewind (see page 1062)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `void rewind(FILE *stream);` **Description** Repositions a file pointer to the beginning of a stream. rewind(stream) is equivalent to fseek(stream, 0L, SEEK_SET), except that rewind clears the end-of-file and error indicators, while fseek clears the end-of-file indicator only. After rewind, the next operation on an update file can be either input or output. **Return Value** None. **Example**
_rmtmp (see page 1063)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int _rmtmp(void);` **Description** Removes temporary files. The _rmtmp function closes and deletes all open temporary file streams which were previously created with tmpfile. The current directory must the same as when the files were created, or the files will not be deleted. **Return Value** _rmtmp returns the total number of temporary files it closed and deleted. **Example**

scanf, wscanf (see page 1064)	**Header File**
	stdio.h
	Category
	Console I/O Routines
	Prototype
	`int scanf(const char *format[, address, ...]);`
	`int wscanf(const wchar_t *format[, address, ...]);`
	Description
	Scans and formats input from the stdin stream.
	Note: For Win32 GUI applications, stdin must be redirected.
	The scanf function:
	• scans a series of input fields one character at a time
	• formats each field according to a corresponding format specifier passed in the format string *format.
	• vsscanf scans and formats input from a string, using an argument list
	There must be one format specifier and address for each input field.
	scanf might stop scanning a particular field before it... more (see page 1064)

setbuf (see page 1066)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `void setbuf(FILE *stream, char *buf);` **Description** Assigns buffering to a stream. setbuf causes the buffer buf to be used for I/O buffering instead of an automatically allocated buffer. It is used after stream has been opened. If buf is null, I/O will be unbuffered; otherwise, it will be fully buffered. The buffer must be BUFSIZ bytes long (specified in stdio.h). stdin and stdout are unbuffered if they are not redirected; otherwise, they are fully buffered. setbuf can be used to change the buffering style used. Unbuffered means that characters written to a stream... more (see page 1066)
setvbuf (see page 1067)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int setvbuf(FILE *stream, char *buf, int type, size_t size);` **Description** Assigns buffering to a stream. setvbuf causes the buffer buf to be used for I/O buffering instead of an automatically allocated buffer. It is used after the given stream is opened. If buf is null, a buffer will be allocated using malloc; the buffer will use size as the amount allocated. The buffer will be automatically freed on close. The size parameter specifies the buffer size and must be greater than zero. The parameter size is limited by the constant UINT_MAX as defined... more (see page 1067)

snprintf;snwprintf (see page 1068)	**Header File**
	stdio.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int snprintf(char* buffer, size_t nsize, const char* fmt, ...);`
	`int snwprintf(wchar_t* buffer, size_t nsize, const wchar_t* fmt, ...);`
	Description
	Sends formatted output to a buffer of a maximum length specified by nsize.
	If the number of bytes to output is:
	• < nsize, then all of the characters have been written, including the terminating '\0' character.
	• == nsize, then nsize characters are written, with no terminating '\0' character.
	> nsize, then only nsize characters are written, with no terminating '\0' character.
	If nsize is 0, then the string will not be... more (see page 1068)

sprintf, swprintf (see page 1068)	**Header File**
	stdio.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int sprintf(char *buffer, const char *format[, argument, ...]);`
	`int swprintf(wchar_t *buffer, const wchar_t *format[, argument, ...]);`
	Description
	Writes formatted output to a string.
	Note: For details on format specifiers, see printf.
	sprintf accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data to a string.
	sprintf applies the first format specifier to the first argument, the second to the second, and so on. There must be the same number of format specifiers as arguments.
	Return... more (see page 1068)

sscanf, swscanf (see page 1069)	**Header File**
	stdio.h
	Category
	Memory and String Manipulation Routines
	Syntax
	`int sscanf(const char *buffer, const char *format[, address, ...]);`
	`int swscanf(const wchar_t *buffer, const wchar_t *format[, address, ...]);`
	Description
	Scans and formats input from a string.
	Note: For details on format specifiers, see scanf.
	sscanf scans a series of input fields, one character at a time, reading from a string. Then each field is formatted according to a format specifier passed to sscanf in the format string pointed to by format. Finally, sscanf stores the formatted input at an address passed to it as an argument following format. There must... more (see page 1069)

stderr, stdin, stdout (see page 1070)	**Header File** stdio.h **Description** Predefined streams automatically opened when the program is started.

_tempnam, _wtempnam (see page 1071)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`char *_tempnam(char *dir, char *prefix)`
	`wchar_t *_wtempnam(wchar_t *dir, wchar_t *prefix)`
	Description
	Creates a unique file name in specified directory.
	The _tempnam function accepts single-byte or multibyte string arguments.
	The _tempnam function creates a unique file name in arbitrary directories. The unique file is not actually created; _tempnam only verifies that it does not currently exist. It attempts to use the following directories, in the order shown, when creating the file name:
	• The directory specified by the TMP environment variable.
	• The dir argument to _tempnam.
	• The P_tmpdir definition in stdio.h. If you edit stdio.h... more (see page 1071)

tmpfile (see page 1072)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `FILE *tmpfile(void);` **Description** Opens a “scratch” file in binary mode. tmpfile creates a temporary binary file and opens it for update (w + b). If you do not change the directory after creating the temporary file, the file is automatically removed when it’s closed or when your program terminates. **Return Value** tmpfile returns a pointer to the stream of the temporary file created. If the file can’t be created, tmpfile returns NULL. **Example**

tmpnam, _wtmpnam (see page 1073)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`char *tmpnam(char *s);`
	`wchar_t *_wtmpnam(wchar_t *s);`
	Description
	Creates a unique file name.
	tmpnam creates a unique file name, which can safely be used as the name of a temporary file. tmpnam generates a different string each time you call it, up to TMP_MAX times. TMP_MAX is defined in stdio.h as 65,535.
	The parameter to tmpnam, s, is either null or a pointer to an array of at least L_tmpnam characters. L_tmpnam is defined in stdio.h. If s is NULL, tmpnam leaves the generated temporary file name in an internal static object and returns... more (see page 1073)

ungetc, ungetwc (see page 1073)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int ungetc(int c, FILE *stream);`
	`wint_t ungetwc(wint_t c, FILE *stream);`
	Description
	Pushes a character back into input stream.
	Note: Do not use this function in Win32 GUI applications.
	ungetc pushes the character c back onto the named input stream, which must be open for reading. This character will be returned on the next call to getc or fread for that stream. One character can be pushed back in all situations. A second call to ungetc without a call to getc will force the previous character to be forgotten. A call to fflush, fseek,... more (see page 1073)

vfprintf, vfwprintf (see page 1074)	**Header File**
	stdio.h
	Category
	Input/output Routines
	Prototype
	`int vfprintf(FILE *stream, const char *format, va_list arglist);`
	`int vfwprintf(FILE *stream, const wchar_t *format, va_list arglist);`
	Description
	Writes formatted output to a stream.
	The v...printf functions are known as alternate entry points for the ...printf functions. They behave exactly like their ...printf counterparts, but they accept a pointer to a list of arguments instead of an argument list.
	For details on format specifiers, see Printf Format Specifiers.
	vfprintf accepts a pointer to a series of arguments, applies to each argument a format specifier contained in the format string pointed to by format, and... more (see page 1074)

vfscanf (see page 1076)	**Header File** stdio.h **Category** Input/output Routines **Prototype** `int vfscanf(FILE *stream, const char *format,va_list arglist);` **Description** Scans and formats input from a stream. The v...scanf functions are known as alternate entry points for the ...scanf functions. They behave exactly like their ...scanf counterparts but they accept a pointer to a list of arguments instead of an argument list. For details on format specifiers, see Scanf Format Specifiers. vfscanf scans a series of input fields one character at a time reading from a stream. Then each field is formatted according to a format specifier passed to vfscanf in the format string pointed... more (see page 1076)

vprintf, vwprintf (see page 1077)	**Header File**
	stdio.h
	Category
	Console I/O Routines
	Prototype
	`int vprintf(const char *format, va_list arglist);`
	`int vwprintf(const wchar_t * format, va_list arglist);`
	Description
	Writes formatted output to stdout.
	Note: Do not use this function in Win32 GUI applications.
	The v...printf functions are known as alternate entry points for the ...printf functions. They behave exactly like their ...printf counterparts, but they accept a pointer to a list of arguments instead of an argument list.
	Note: For details on format specifiers, see Printf Format Specifiers.
	vprintf accepts a pointer to a series of arguments, applies to each a format specifier contained in the... more (see page 1077)

vscanf (see page 1078)	**Header File** stdio.h **Category** Console I/O Routines **Prototype** `int vscanf(const char *format, va_list arglist);` **Description** Scans and formats input from stdin. **Note:** Do not use this function in Win32 GUI applications. The v...scanf functions are known as alternate entry points for the ...scanf functions. They behave exactly like their ...scanf counterparts, but they accept a pointer to a list of arguments instead of an argument list. **Note:** For details on format specifiers, see Scanf Format Specifiers. vscanf scans a series of input fields, one character at a time, reading from stdin. Then each field is formatted according to a format... more (see page 1078)

vsnprintf;vsnwprintf (see page 1079)	**Header File**
	stdio.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int vsnprintf(char* buffer, size_t nsize, const char* format, va_list param);`
	`int vsnwprintf(wchar_t* buffer, size_t nsize, const wchar_t* format, va_list param);`
	Description
	Sends formatted output to a buffer of maximum length specified by nsize.
	If the number of bytes to output is:
	• < nsize, then all of the characters have been written, including the terminating '\0' character.
	• == nsize, then nsize characters are written, with no terminating '\0' character.
	> nsize, then only nsize characters are written, with no terminating '\0' character.
	If nsize is 0, then the string will not... more (see page 1079)

vsprintf, vswprintf (see page 1079)	**Header File**
	stdio.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int vsprintf(char *buffer, const char *format, va_list arglist);`
	`int vswprintf(wchar_t *buffer, const wchar_t *format, va_list arglist);`
	Description
	Writes formatted output to a string.
	The v...printf functions are known as alternate entry points for the ...printf functions. They behave exactly like their ...printf counterparts, but they accept a pointer to a list of arguments instead of an argument list.
	vsprintf accepts a pointer to a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data to a string.... more (see page 1079)

3.1.4.28.1 BUFSIZ #define

Header File

stdio.h

Description

Default buffer size used by setbuf function.

3.1.4.28.2 _F_xxxx #defines

Header File

stdio.h

Description

File status flags of streams

Name	Meaning
_F_RDWR	Read and write

_F_READ	Read-only file
_F_WRIT	Write-only file
_F_BUF	Malloc'ed buffer data
_F_LBUF	Line-buffered file
_F_ERR	Error indicator
_F_EOF	EOF indicator
_F_BIN	Binary file indicator
_F_IN	Data is incoming
_F_OUT	Data is outgoing
_F_TERM	File is a terminal

3.1.4.28.3 OPEN_MAX #define

Header File

stdio.h

Description

Number of files that can be open simultaneously.

Name	Meaning
FOPEN_MAX	Maximum files per process
SYS_OPEN	Maximum files for system

3.1.4.28.4 L_ctermid #define

Header File

stdio.h

Description

The length of a device id string.

3.1.4.28.5 L_tmpnam #define

Header File

stdio.h

Description

Size of an array large enough to hold a temporary file name string.

3.1.4.28.6 TMP_MAX #define

Header File

stdio.h

Description

Maximum number of unique file names.

3.1.4.28.7 _IOxxx #defines

Header File

stdio.h

Description

Constants for defining buffering style to be used with a file.

Name	Meaning
_IOFBF	The file is fully buffered. When a buffer is empty, the next input operation will attempt to fill the entire buffer.
	On output, the buffer will be completely filled before any data is written to the file.
_IOLBF	The file is line buffered. When a buffer is empty, the next input operation will still attempt to fill the entire buffer.
	On output, however, the buffer will be flushed whenever a newline character is written to the file.
_IONBF	The file is unbuffered. The buf and size parameters of setbuf are ignored. Each input operation will read directly from the file, and each output operation will immediately write the data to the file.

3.1.4.28.8 _fsopen, _wfsopen

Header File

stdio.h, share.h

Category

Input/output Routines

Prototype

```
FILE *_fsopen(const char *filename, const char *mode, int shflag);
FILE *_wfsopen(const wchar_t *filename, const wchar_t *mode, int shflag);
```

Description

Opens a stream with file sharing.

_fsopen opens the file named by filename and associates a stream with it. _fsopen returns a pointer that is used to identify the stream in subsequent operations.

The mode string used in calls to _fsopen is one of the following values:

r	Open for reading only.
w	Create for writing. If a file by that name already exists, it will be overwritten.
a	Append; open for writing at end of file. or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).
w+	Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.

a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode append a t to the mode string (rt w+t and so on). Similarly to specify binary mode append a b to the mode string (wb a+b and so on). _fsopen also allows the t or b to be inserted between the letter and the + character in the mode string; for example rt+ is equivalent to r+t. If a t or b is not given in the mode string the mode is governed by the global variable _fmode. If _fmode is set to O_BINARY files are opened in binary mode. If _fmode is set to O_TEXT they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream, however:

- output cannot be directly followed by input without an intervening fseek or rewind
- input cannot be directly followed by output without an intervening fseek, rewind, or an input that encounters end-of-file

shflag specifies the type of file-sharing allowed on the file filename. Symbolic constants for shflag are defined in share.h.

SH_COMPAT	Sets compatibility mode
SH_DENYRW	Denies read/write access
SH_DENYWR	Denies write access
SH_DENYRD	Denies read access
SH_DENYNONE	Permits read/write access
SH_DENYNO	Permits read/write access

Return Value

On successful completion _fsopen returns a pointer to the newly opened stream.

On error it returns NULL.

Example

```
#include <io.h>
#include <process.h>
#include <share.h>
#include <stdio.h>
int main(void)
{
  FILE *f;
  int status;
  f = _fsopen("TESTFILE.DAT", "r", SH_DENYNO);
  if (f == NULL)
  {
    printf("_fsopen failed\n");
    exit(1);
  }
  status = access("TESTFILE.DAT", 6);
  if (status == 0)
    printf("read/write access allowed\n");
  else
    printf("read/write access not allowed\n");
  fclose(f);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fsopen		+		
_wfsopen		NT only		

3.1.4.28.9 _pclose

Header File

stdio.h

Category

Input/output Routines, Process Control Routines

Prototype

```
int _pclose(FILE * stream);
```

Description

Waits for piped command to complete.

_pclose closes a pipe stream created by a previous call to _popen, and then waits for the associated child command to complete.

Return Value

On success, _pclose returns the termination status of the child command. This is the same value as the termination status returned by cwait, except that the high and low order bytes of the low word are swapped.

On error, it returns -1.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.28.10 _popen, _wpopen

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *_popen (const char *command, const char *mode);
FILE *_wpopen (const wchar_t *command, const wchar_t *mode);
```

Description

Creates a command processor pipe.

The _popen function creates a pipe to the command processor. The command processor is executed asynchronously, and is passed the command line in command. The mode string specifies whether the pipe is connected to the command processor's standard input or output, and whether the pipe is to be opened in binary or text mode.

The mode string can take one of the following values:

rt	Read child command's standard output (text).
rb	Read child command's standard output (binary).

wt	Write to child command's standard input (text).
wb	Write to child command's standard input (binary).

The terminating t or b is optional; if missing, the translation mode is determined by the external variable _fmode.

Use the _pclose function to close the pipe and obtain the return code of the command.

Return Value

On success, _popen returns a FILE pointer that can be used to read the standard output of the command, or to write to the standard input of the command, depending on the mode string.

On error, it returns NULL.

Example

```
/* this program initiates a child process to run the dir command
   and pipes the directory listing from the child to the parent.
*/
#include <stdio.h>     // popen() pclose() feof() fgets() puts()
#include <string.h>    // strlen()
int main( )
{
  FILE* handle;        // handle to one end of pipe
  char message[256];   // buffer for text passed through pipe
  int status;          // function return value
  // open a pipe to receive text from a process running "DIR"
  handle = _popen("dir /b", "rt");
  if (handle == NULL)
  {
    perror("_popen error");
  }
  // read and display input received from the child process
  while (fgets(message, sizeof(message), handle))
  {
    fprintf(stdout, message);
  }
  // close the pipe and check the return status
    status = _pclose(handle);
  if (status == -1)
  {
    perror("_pclose error");
  }
  return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_popen		+		
_wpopen		NT only		

3.1.4.28.11 _snprintf;_snwprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Syntax

```
int _snprintf(char* buffer, size_t nsize, const char* format, ...);
int _snwprintf(wchar_t* buffer, size_t nsize, const wchar_t* format, ...);
```

Description

Sends formatted output to a string of a maximum length specified by nsize. _snprintf and _snwprintf are Microsoft compatible with the _snprintf and _snprintfw functions, respectively.

If the number of bytes to output is:

- < nsize, then all of the characters have been written, including the terminating '\0' character.
- == nsize, then nsize characters are written with no terminating '\0' character.

If nsize is 0, then the string will not be written to (and may be NULL).

If nsize is too small, then return value is -1, and only nsize characters are written, with no terminating '\0' character.

Return Value

Number of bytes output or –1 if nsize is too small.

3.1.4.28.12 _vsnprintf;_vsnwprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int _vsnprintf(char* buffer, size_t nsize, const char* format, va_list param);
int _vsnwprintf(wchar_t* buffer, size_t nsize, const wchar_t* format, va_list param);
```

Description

Sends formatted output to a string of a maximum length specified by nsize. _vsnprintf and _vsnwprintf are Microsoft compatible with the _vsnprintf and _vsnprintfw functions, respectively.

If the number of bytes to output is:

- < nsize, then all of the characters have been written, including the terminating '\0' character.
- == nsize, then nsize characters are written with no terminating '\0' character.

If nsize is 0, then the string will not be written to (and may be NULL).

If nsize is too small, then return value is -1, and only nsize characters are written, with no terminating '\0' character.

Return Value

Number of bytes output or –1 if nsize is too small.

3.1.4.28.13 clearerr

Header File

stdio.h

Category

Input/output Routines

Prototype

```
void clearerr(FILE *stream);
```

Description

Resets error indication.

clearerr resets the named stream's error and end-of-file indicators to 0. Once the error indicator is set, stream operations continue to return error status until a call is made to clearerr or rewind. The end-of-file indicator is reset with each input operation.

Return Value

None.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *fp;
   char ch;
   /* open a file for writing */
   fp = fopen("DUMMY.FIL", "w");
   /* force an error condition by attempting to read */
   ch = fgetc(fp);
   printf("%c\n",ch);
   if (ferror(fp))
   {
      /* display an error message */
      printf("Error reading from DUMMY.FIL\n");
      /* reset the error and EOF indicators */
      clearerr(fp);
   }
   fclose(fp);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.14 EOF #define

Header File

stdio.h

Description

A constant indicating that end-of-file has been reached on a file.

3.1.4.28.15 fclose

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fclose(FILE *stream);
```

Description

Closes a stream.

fclose closes the named stream. All buffers associated with the stream are flushed before closing. System-allocated buffers are freed upon closing. Buffers assigned with setbuf or setvbuf are not automatically freed. (But if setvbuf is passed null for the buffer pointer it will free it upon close.)

Return Value

fclose returns 0 on success. It returns EOF if any errors were detected.

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.16 _fcloseall

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _fcloseall(void);
```

Description

Closes open streams.

_fcloseall closes all open streams except

stdauxstdstreams

When _fcloseall flushes the associated buffers before closing a stream. The buffers allocated by the system are released.

Note: stdprn and stdaux streams are not available in Win32.

Return Value

_fcloseall returns the total number of streams it closed. The _fcloseall function returns EOF if any errors were detected.

Example

```
#include <stdio.h>
int main(void)
{
   int streams_closed;
   /* open two streams */
   fopen("DUMMY.ONE", "w");
   fopen("DUMMY.TWO", "w");
   /* close the open streams */
   streams_closed = fcloseall();
```

```
    if (streams_closed == EOF)
       /* issue an error message */
       perror("Error");
    else
       /* print result of fcloseall() function */
       printf("%d streams were closed.\n", streams_closed);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.28.17 _fdopen, _wfdopen

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *_fdopen(int handle, char *type);
```

```
FILE *_wfdopen(int handle, wchar_t *type);
```

Description

Associates a stream with a file handle.

_fdopen associates a stream with a file handle obtained from creat, dup, dup2, or open.

The type of stream must match the mode of the open handle.

The type string used in a call to _fdopen is one of the following values:

r	Open for reading only. _fdopen returns NULL if the file cannot be opened.
w	Create for writing. If the file already exists, its contents are overwritten.
a	Append; open for writing at end-of-file or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing). _fdopen returns NULL if the file cannot be opened.
w+	Create a new file for update. If the file already exists, its contents are overwritten.
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode, append t to the value of the type string (for example, rt or w+t).

Similarly, to specify binary mode, append b to the type string (for example, rb or w+b).

If t or b is not given in the type string, the mode is governed by the global variable _fmode.

If _fmode is set to O_BINARY, files will be opened in binary mode.

If _fmode is set to O_TEXT, files will be opened in text mode.

Note: The O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,

- output cannot be directly followed by input without an intervening fseek or rewind
- input cannot be directly followed by output without an intervening fseek, rewind, or an input that encounters end-of-file

Return Value

On successful completion _fdopen returns a pointer to the newly opened stream. In the event of error it returns NULL.

Example

```
#include <sys\stat.h>
#include <stdio.h>
#include <fcntl.h>
#include <io.h>
int main(void)
{
   int handle;
   FILE *stream;
   /* open a file */
   handle = open("DUMMY.FIL", O_CREAT,
                 S_IREAD | S_IWRITE);
   /* now turn the handle into a stream */
   stream = fdopen(handle, "w");
   if (stream == NULL)
      printf("fdopen failed\n");
   else
   {
      fprintf(stream, "Hello world\n");
      fclose(stream);
   }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fdopen	+	+		
_wfdopen		+		

3.1.4.28.18 feof

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int feof(FILE *stream);
```

Description

Detects end-of-file on a stream.

feof is a macro that tests the given stream for an end-of-file indicator. Once the indicator is set read operations on the file return the indicator until rewind is called or the file is closed. The end-of-file indicator is reset with each input operation.

Return Value

feof returns nonzero if an end-of-file indicator was detected on the last input operation on the named stream and 0 if end-of-file

has not been reached.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   /* open a file for reading */
   stream = fopen("DUMMY.FIL", "r");
   /* read a character from the file */
   fgetc(stream);
   /* check for EOF */
   if (feof(stream))
      printf("We have reached end-of-file\n");
   /* close the file */
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.19 ferror

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int ferror(FILE *stream);
```

Description

Detects errors on stream.

ferror is a macro that tests the given stream for a read or write error. If the stream's error indicator has been set it remains set until clearerr or rewind is called or until the stream is closed.

Return Value

ferror returns nonzero if an error was detected on the named stream.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   /* open a file for writing */
   stream = fopen("DUMMY.FIL", "w");
   /* force an error condition by attempting to read */
   (void) getc(stream);
   if (ferror(stream))  /* test for an error on the stream */
   {
      /* display an error message */
      printf("Error reading from DUMMY.FIL\n");
```

```
      /* reset the error and EOF indicators */
      clearerr(stream);
   }
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.20 fflush

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fflush(FILE *stream);
```

Description

Flushes a stream.

If the given stream has buffered output fflush writes the output for stream to the associated file.

The stream remains open after fflush has executed. fflush has no effect on an unbuffered stream.

Return Value

fflush returns 0 on success. It returns EOF if any errors were detected.

Example

```
#include <string.h>
#include <stdio.h>
#include <io.h>
void flush(FILE *stream);
int main(void)
{
   FILE *stream;
   char msg[] = "This is a test";
   /* create a file */
   stream = fopen("DUMMY.FIL", "w");
   /* write some data to the file */
   fwrite(msg, strlen(msg), 1, stream);
   printf("Press ENTER to flush DUMMY.FIL:");
   getchar();
   /* flush the data to DUMMY.FIL without closing it */
   flush(stream);
   printf("\nFile was flushed, Press ENTER to quit:");
   getchar();
   return 0;
}
void flush(FILE *stream)
{
     int duphandle;
     /* flush the stream's internal buffer */
```

```
    fflush(stream);
    /* make a duplicate file handle */
    duphandle = dup(fileno(stream));
    /* close the duplicate handle to flush the DOS buffer */
    close(duphandle);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.21 fgetc, fgetwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fgetc(FILE *stream);
```

```
wint_t fgetwc(FILE *stream);
```

Description

Gets character from stream.

fgetc returns the next character on the named input stream.

Return Value

On success fgetc returns the character read after converting it to an **int** without sign extension. On end-of-file or error it returns EOF.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   FILE *stream;
   char string[] = "This is a test";
   char ch;
   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");
   /* write a string into the file */
   fwrite(string, strlen(string), 1, stream);
   /* seek to the beginning of the file */
   fseek(stream, 0, SEEK_SET);
   do
   {
      /* read a char from the file */
      ch = fgetc(stream);
      /* display the character */
      putchar(ch);
   } while (ch != EOF);
   fclose(stream);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fgetc	+	+	+	+
fgetwc		+	+	+

3.1.4.28.22 _fgetchar, _fgetwchar

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int _fgetchar(void);
```

```
wint_t _fgetwchar(void);
```

Description

Reads a character from stdin.

_fgetchar returns the next character from stdin. It is defined as fgetc(stdin).

Note: For Win32 GUI applications, stdin must be redirected.

Return Value

On success _fgetchar returns the character read after converting it to an **int** without sign extension. On end-of-file or error it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char ch;
   /* prompt the user for input */
   printf("Enter a character followed by <Enter>: ");
   /* read the character from stdin */
   ch = fgetchar();
   /* display what was read */
   printf("The character read is: '%c'\n", ch);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fgetchar		+		
_fgetwchar		+		

3.1.4.28.23 fgetpos

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fgetpos(FILE *stream, fpos_t *pos);
```

Description

Gets the current file pointer.

fgetpos stores the position of the file pointer associated with the given stream in the location pointed to by pos. The exact value is unimportant; its value is opaque except as a parameter to subsequent fsetpos calls.

Return Value

On success fgetpos returns 0. On failure it returns a nonzero value and sets the global variable errno to

EBADF	Bad file number
EINVAL	Invalid number

Example

```
#include <stdlib.h>
#include <stdio.h>
void showpos(FILE *stream);
int main(void)
{
   FILE *stream;
   fpos_t filepos;
   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");
   /* save the file pointer position */
   fgetpos(stream, &filepos);
   /* write some data to the file */
   fprintf(stream, "This is a test");
   /* show the current file position */
   showpos(stream);
   /* set a new file position, display it */
   if (fsetpos(stream, &filepos) == 0)
     showpos(stream);
   else
   {
      fprintf(stderr, "Error setting file pointer.\n");
      exit(1);
   }
   /* close the file */
   fclose(stream);
   return 0;
}
void showpos(FILE *stream)
{
   fpos_t pos;
   /* display the current file pointer
      position of a stream */
   fgetpos(stream, &pos);
   printf("File position: %ld\n", pos);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.24 fgets, fgetws

Header File

stdio.h

Category

Input/output Routines

Prototype

```
char *fgets(char *s, int n, FILE *stream);
```

```
wchar_t *fgetws(wchar_t *s, int n, FILE *stream); // Unicode version
```

Description

Gets a string from a stream.

fgets reads characters from stream into the string s. The function stops reading when it reads either n - 1 characters or a newline character whichever comes first. fgets retains the newline character at the end of s. A null byte is appended to s to mark the end of the string.

Return Value

On success fgets returns the string pointed to by s; it returns NULL on end-of-file or error.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   FILE *stream;
   char string[] = "This is a test";
   char msg[20];
   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");
   /* write a string into the file */
   fwrite(string, strlen(string), 1, stream);
   /* seek to the start of the file */
   fseek(stream, 0, SEEK_SET);
   /* read a string from the file */
   fgets(msg, strlen(string)+1, stream);
   /* display the string */
   printf("%s", msg);
   fclose(stream);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fgets	+	+	+	+
fgetws		+	+	+

3.1.4.28.25 _fileno

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _fileno(FILE *stream);
```

Description

Gets the file handle.

_fileno is a macro that returns the file handle for the given stream. If stream has more than one handle _fileno returns the handle assigned to the stream when it was first opened.

Return Value

_fileno returns the integer file handle associated with stream.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   int handle;
   /* create a file */
   stream = fopen("DUMMY.FIL", "w");
   /* obtain the file handle associated with the stream */
   handle = fileno(stream);
   /* display the handle number */
   printf("handle number: %d\n", handle);
   /* close the file */
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.28.26 _flushall

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _flushall(void);
```

Description

Flushes all streams.

_flushall clears all buffers associated with open input streams and writes all buffers associated with open output streams to their respective files. Any read operation following _flushall reads new data into the buffers from the input files. Streams stay open after _flushall executes.

Return Value

_flushall returns an integer the number of open input and output streams.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   /* create a file */
   stream = fopen("DUMMY.FIL", "w");
   /* flush all open streams */
   printf("%d streams were flushed.\n", flushall());
   /* close the file */
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.28.27 fopen, _wfopen

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *fopen(const char *filename, const char *mode);
```

```
FILE *_wfopen(const wchar_t *filename, const wchar_t *mode);
```

Description

Opens a stream.

fopen opens the file named by filename and associates a stream with it. fopen returns a pointer to be used to identify the stream in subsequent operations.

The mode string used in calls to fopen is one of the following values:

r	Open for reading only.
w	Create for writing. If a file by that name already exists, it will be overwritten.
a	Append; open for writing at end-of-file or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).

w+	Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten.
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode append a t to the mode string (rt w+t and so on). Similarly to specify binary mode append a b to the mode string (wb a+b and so on). fopen also allows the t or b to be inserted between the letter and the + character in the mode string; for example rt+ is equivalent to r+t.

If a t or b is not given in the mode string the mode is governed by the global variable _fmode. If _fmode is set to O_B/INARY files are opened in binary mode. If _fmode is set to O_TEXT they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,

- output cannot be directly followed by input without an intervening fseek or rewind
- input cannot be directly followed by output without an intervening fseek, rewind, or an input that encounters end-of-file

Return Value

On successful completion fopen returns a pointer to the newly opened stream. In the event of error it returns NULL.

Portability

	POSIX	Win32	ANSI C	ANSI C++
fopen	+	+	+	+
_wfopen		NT only		

3.1.4.28.28 fprintf, fwprintf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fprintf(FILE *stream, const char *format[, argument, ...]);
```

```
int fwprintf(FILE *stream, const wchar_t *format[, argument, ...]);
```

Description

Writes formatted output to a stream.

fprintf accepts a series of arguments applies to each a format specifier contained in the format string pointed to by format and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.

Note: For details on format specifiers, see printf Format Specifiers.

Return Value

fprintf returns the number of bytes output. In the event of error it returns EOF.

Example

```
#include <stdio.h>
int main(void)
```

```
{
   FILE *stream;
   int i = 100;
   char c = 'C';
   float f = 1.234;
   /* open a file for update */
   stream = fopen("DUMMY.FIL", "w+");
   /* write some data to the file */
   fprintf(stream, "%d %c %f", i, c, f);
   /* close the file */
   fclose(stream);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fprintf	+	+	+	+
fwprintf		+	+	+

3.1.4.28.29 fputc, fputwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fputc(int c, FILE *stream);
```

```
wint_t fputwc(wint_t c, FILE *stream);
```

Description

Puts a character on a stream.

fputc outputs character c to the named stream.

Note: For Win32 GUI applications, stdin must be redirected.

Return Value

On success, fputc returns the character c. On error, it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char msg[] = "Hello world";
   int i = 0;
   while (msg[i])
   {
      fputc(msg[i], stdout);
      i++;
   }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fputc	+	+	+	+
fputwc		+	+	+

3.1.4.28.30 _fputchar, _fputwchar

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _fputchar(int c);
wint_t _fputwchar(wint_t c);
```

Description

Outputs a character to stdout.

_fputchar outputs character c to stdout. _fputchar(c) is the same as fputc(cstdout).

Note: For Win32 GUI applications, stdout must be redirected.

Return Value

On success _fputchar returns the character c.

On error it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char msg[] = "This is a test";
   int i = 0;
   while (msg[i])
   {
      fputchar(msg[i]);
      i++;
   }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fputchar		+		
_fputwchar		+		

3.1.4.28.31 fputs, fputws

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fputs(const char *s, FILE *stream);
```

```
int fputws(const wchar_t *s, FILE *stream);
```

Description

Outputs a string on a stream.

fputs copies the null-terminated string s to the given output stream; it does not append a newline character and the terminating null character is not copied.

Return Value

On success fputs returns a non-negative value.

On error it returns a value of EOF.

Example

```
#include <stdio.h>
int main(void)
{
   /* write a string to standard output */
   fputs("Hello world\n", stdout);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
fputs	+	+	+	+
fputws		+	+	+

3.1.4.28.32 fread

Header File

stdio.h

Category

Input/output Routines

Prototype

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Description

Reads data from a stream.

fread reads n items of data each of length size bytes from the given input stream into a block pointed to by ptr.

The total number of bytes read is (n * size).

Return Value

On success fread returns the number of items (not bytes) actually read.

On end-of-file or error it returns a short count (possibly 0).

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   FILE *stream;
   char msg[] = "this is a test";
   char buf[20];
   if ((stream = fopen("DUMMY.FIL", "w+"))
       == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }
   /* write some data to the file */
   fwrite(msg, strlen(msg)+1, 1, stream);
   /* seek to the beginning of the file */
   fseek(stream, SEEK_SET, 0);
   /* read the data and display it */
   fread(buf, strlen(msg)+1, 1, stream);
   printf("%s\n", buf);
   fclose(stream);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.33 freopen, _wfreopen

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

```
FILE *_wfreopen(const wchar_t *filename, const wchar_t *mode, FILE *stream);
```

Description

Associates a new file with an open stream.

freopen substitutes the named file in place of the open stream. It closes stream regardless of whether the open succeeds. freopen is useful for changing the file attached to stdin, stdout, or stderr.

The mode string used in calls to fopen is one of the following values:

r	Open for reading only.
w	Create for writing. .

a	Append; open for writing at end-of-file or create for writing if the file does not exist.
r+	Open an existing file for update (reading and writing).
w+	Create a new file for update (reading and writing).
a+	Open for append; open (or create if the file does not exist) for update at the end of the file.

To specify that a given file is being opened or created in text mode append a t to the mode string (rt w+t and so on); similarly to specify binary mode append a b to the mode string (wb a+b and so on).

If a t or b is not given in the mode string the mode is governed by the global variable _fmode. If _fmode is set to O_BINARY files are opened in binary mode. If _fmode is set to O_TEXT they are opened in text mode. These O_... constants are defined in fcntl.h.

When a file is opened for update, both input and output can be done on the resulting stream; however,

- output cannot be directly followed by input without an intervening fseek or rewind
- input cannot be directly followed by output without an intervening fseek, rewind, or an input that encounters end-of-file

Return Value

On successful completion freopen returns the argument stream.

On error it returns NULL.

Example

```
#include <stdio.h>
int main(void)
{
   /* redirect standard output to a file */
   if (freopen("OUTPUT.FIL", "w", stdout)
       == NULL)
      fprintf(stderr, "error redirecting stdout\n");
   /* this output will go to a file */
   printf("This will go into a file.");
   /* close the standard output stream */
   fclose(stdout);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
freopen	+	+	+	+
_wfreopen		NT only		

3.1.4.28.34 fscanf, fwscanf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fscanf(FILE *stream, const char *format[, address, ...]);
```

```
int fwscanf(FILE *stream, const wchar_t *format[, address, ...]);
```

Description

Scans and formats input from a stream.

fscanf scans a series of input fields one character at a time reading from a stream. Then each field is formatted according to a format specifier passed to fscanf in the format string pointed to by format. Finally fscanf stores the formatted input at an address passed to it as an argument following format. The number of format specifiers and addresses must be the same as the number of input fields.

Note: For details on format specifiers, see scanf Format Specifiers.

fscanf can stop scanning a particular field before it reaches the normal end-of-field character (whitespace) or it can terminate entirely for a number of reasons. See scanf for a discussion of possible causes.

Return Value

fscanf returns the number of input fields successfully scanned, converted and stored. The return value does not include scanned fields that were not stored.

If fscanf attempts to read at end-of-file, the return value is EOF. If no fields were stored, the return value is 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   int i;
   printf("Input an integer: ");
   /* read an integer from the
      standard input stream */
   if (fscanf(stdin, "%d", &i))
      printf("The integer read was: %i\n", i);
   else
   {
      fprintf(stderr, "Error reading an integer from stdin.\n");
      exit(1);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.35 fseek

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fseek(FILE *stream, long offset, int whence);
```

Description

Repositions a file pointer on a stream.

fseek sets the file pointer associated with stream to a new position that is offset bytes from the file location given by whence. For text mode streams offset should be 0 or a value returned by ftell.

whence must be one of the values 0. 1, or 2 which represent three symbolic constants (defined in stdio.h) as follows:

fseek discards any character pushed back using ungetc. fseek is used with stream I/O; for file handle I/O use lseek.

After fseek the next operation on an update file can be either input or output.

Return Value

fseek returns 0 if the pointer is successfully moved and nonzero on failure.

fseek might return a 0 indicating that the pointer has been moved successfully when in fact it has not been. This is because DOS, which actually resets the pointer, does not verify the setting. fseek returns an error code only on an unopened file or device.

In the event of an error return the global variable errno is set to one of the following values:

EBADF	Bad file pointer
EINVAL	Invalid argument
ESPIPE	Illegal seek on device

Example

```
#include <stdio.h>
long filesize(FILE *stream);
int main(void)
{
   FILE *stream;
   stream = fopen("MYFILE.TXT", "w+");
   fprintf(stream, "This is a test");
   printf("Filesize of MYFILE.TXT is %ld bytes\n", filesize(stream));
   fclose(stream);
   return 0;
}
long filesize(FILE *stream)
{
   long curpos, length;
   curpos = ftell(stream);
   fseek(stream, 0L, SEEK_END);
   length = ftell(stream);
   fseek(stream, curpos, SEEK_SET);
   return length;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.36 fsetpos

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

Positions the file pointer of a stream.

fsetpos sets the file pointer associated with stream to a new position. The new position is the value obtained by a previous call to fgetpos on that stream. It also clears the end-of-file indicator on the file that stream points to and undoes any effects of ungetc on that file. After a call to fsetpos the next operation on the file can be input or output.

Return Value

On success fsetpos returns 0.

On failure it returns a nonzero value and also sets the global variable errno to a nonzero value.

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.37 ftell

Header File

stdio.h

Category

Input/output Routines

Prototype

```
long int ftell(FILE *stream);
```

Description

Returns the current file pointer.

ftell returns the current file pointer for stream. The offset is measured in bytes from the beginning of the file (if the file is binary). The value returned by ftell can be used in a subsequent call to fseek.

Return Value

ftell returns the current file pointer position on success. It returns -1L on error and sets the global variable errno to a positive value.

In the event of an error return the global variable errno is set to one of the following values:

EBADF	Bad file pointer
ESPIPE	Illegal seek on device

Example

```
#include <stdio.h>
int main(void)
{
   FILE *stream;
   stream = fopen("MYFILE.TXT", "w+");
   fprintf(stream, "This is a test");
   printf("The file pointer is at byte %ld\n", ftell(stream));
   fclose(stream);
```

```
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.38 fwrite

Header File

stdio.h

Category

Input/output Routines

Prototype

```
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
```

Description

Writes to a stream.

fwrite appends n items of data each of length size bytes to the given output file. The data written begins at ptr. The total number of bytes written is (n x size). ptr in the declarations is a pointer to any object.

Return Value

On successful completion fwrite returns the number of items (not bytes) actually written.

On error it returns a short count.

Example

```
#include <stdio.h>
struct mystruct
{
  int i;
  char ch;
};
int main(void)
{
   FILE *stream;
   struct mystruct s;
   if ((stream = fopen("TEST.$$$", "wb")) == NULL) /* open file TEST.$$$ */
   {
      fprintf(stderr, "Cannot open output file.\n");
      return 1;
   }
   s.i = 0;
   s.ch = 'A';
   fwrite(&s, sizeof(s), 1, stream); /* write struct s to file */
   fclose(stream); /* close file */
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.39 getc, getwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int getc(FILE *stream);
```

```
wint_t getwc(FILE *stream);
```

Description

Gets character from stream.

getc returns the next character on the given input stream and increments the stream's file pointer to point to the next character.

Note: For Win32 GUI applications, stdin must be redirected.

Return Value

On success, getc returns the character read, after converting it to an **int** without sign extension.

On end-of-file or error, it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char ch;
   printf("Input a character:");
/* read a character from the
standard input stream */
   ch = getc(stdin);
   printf("The character input was: '%c'\n", ch);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getc	+	+	+	+
getwc		+	+	+

3.1.4.28.40 getchar, getwchar

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int getchar(void);
wint_t getwchar(void);
```

Description

Gets character from stdin.

getchar is a macro that returns the next character on the named input stream stdin. It is defined to be getc(stdin).

Note: Do not use this function in Win32 GUI applications.

Return Value

On success, getchar returns the character read, after converting it to an **int** without sign extension.

On end-of-file or error, it returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   int c;
/*
Note that getchar reads from stdin and is line buffered; this means it will not return until
you press ENTER.
 */
   while ((c = getchar()) != '\n')
      printf("%c", c);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getchar	+	+	+	+
getwchar		+	+	+

3.1.4.28.41 gets, _getws

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
char *gets(char *s);
wchar_t *_getws(wchar_t *s); // Unicode version
```

Description

Gets a string from stdin.

gets collects a string of characters terminated by a new line from the standard input stream stdin and puts it into s. The new line is replaced by a null character (\0) in s.

gets allows input strings to contain certain whitespace characters (spaces, tabs). gets returns when it encounters a new line; everything up to the new line is copied into s.

The gets function is not length-terminated. If the input string is sufficiently large, data can be overwritten and corrupted. The fgets function provides better control of input strings.

Note: For Win32 GUI applications, stdin must be redirected.

Return Value

On success, gets returns the string argument s.

On end-of-file or error, it returns NULL

Example

```
#include <stdio.h>
int main(void)
{
   char string[80];
   printf("Input a string:");
   gets(string);
   printf("The string input was: %s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
gets	+	+	+	+
_getws		+		

3.1.4.28.42 _getw

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _getw(FILE *stream);
```

Description

Gets an integer from stream.

_getw returns the next integer in the named input stream. It assumes no special alignment in the file.

_getw should not be used when the stream is opened in text mode.

Return Value

_getw returns the next integer on the input stream.

On end-of-file or error, _getw returns EOF.

Note: Because EOF is a legitimate value for _getw to return, feof or ferror should be used to detect end-of-file or error.

Example

```
#include <stdio.h>
#include <stdlib.h>
#define FNAME "test.$$$"
```

```
int main(void)
{
FILE *fp;
int word;
/* place the word in a file */
fp = fopen(FNAME, "wb");
if (fp == NULL)
{
printf("Error opening file %s\n", FNAME);
      exit(1);
   }
   word = 94;
   putw(word,fp);
   if (ferror(fp))
       printf("Error writing to file\n");
   else
       printf("Successful write\n");
   fclose(fp);
   /* reopen the file */
   fp = fopen(FNAME, "rb");
   if (fp == NULL)
   {
      printf("Error opening file %s\n", FNAME);
      exit(1);
   }
   /* extract the word */
   word = getw(fp);
   if (ferror(fp))
       printf("Error reading file\n");
   else
       printf("Successful read: word = %d\n", word);
   /* clean up */
   fclose(fp);
   unlink(FNAME);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.28.43 printf, wprintf

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int printf(const char *format[, argument, ...]);
```

```
int wprintf(const wchar_t *format[, argument, ...]);
```

Description

Writes formatted output to stdout.

The printf function:

- Accepts a series of arguments
- Applies to each argument a format specifier contained in the format string *format
- Outputs the formatted data (to the screen, a stream, stdout, or a string)

There must be enough arguments for the format. If there are not, the results will be unpredictable and likely disastrous. Excess arguments (more than required by the format) are merely ignored.

Note: For Win32 GUI applications, stdout must be redirected.

Return Value

On success, printf returns the number of bytes output.

On error, printf returns EOF.

More About printf

Example

```
#include <stdio.h>
#include <string.h>
#define I 555
#define R 5.5
int main(void)
{
   int i,j,k,l;
   char buf[7];
   char *prefix = buf;
   char tp[20];
   printf("prefix  6d      6o      8x        10.2e        "
          "10.2f\n");
   strcpy(prefix,"%");
   for (i = 0; i < 2; i++)
   {
      for (j = 0; j < 2; j++)
         for (k = 0; k < 2; k++)
            for (l = 0; l < 2; l++)
            {
               if (i==0)  strcat(prefix,"-");
               if (j==0)  strcat(prefix,"+");
               if (k==0)  strcat(prefix,"#");
               if (l==0)  strcat(prefix,"0");
               printf("%5s |",prefix);
               strcpy(tp,prefix);
               strcat(tp,"6d |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"6o |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"8x |");
               printf(tp,I);
               strcpy(tp,"");
               strcpy(tp,prefix);
               strcat(tp,"10.2e |");
               printf(tp,R);
               strcpy(tp,prefix);
               strcat(tp,"10.2f |");
               printf(tp,R);
               printf("  \n");
               strcpy(prefix,"%");
            }
       }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
printf	+	+	+	+
_wprintf		+		

3.1.4.28.44 putc, putwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int putc(int c, FILE *stream);
wint_t putwc(wint_t c, FILE *stream);
```

Description

Outputs a character to a stream.

putc is a macro that outputs the character c to the stream given by stream.

Return Value

On success, putc returns the character printed, c.

On error, putc returns EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char msg[] = "Hello world\n";
   int i = 0;
   while (msg[i])
      putc(msg[i++], stdout);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
putc	+	+	+	+
putwc		+	+	+

3.1.4.28.45 putchar, putwchar

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int putchar(int c);

wint_t putwchar(wint_t c);
```

Description

putchar is a macro defined to be putc(c, stdout).

Note: For Win32 GUI applications, stdout must be redirected.

Return Value

On success, putchar returns the character c. On error, putchar returns EOF.

Example

```
#include <stdio.h>
/* define some box-drawing characters */
#define LEFT_TOP  0xDA
#define RIGHT_TOP 0xBF
#define HORIZ     0xC4
#define VERT      0xB3
#define LEFT_BOT  0xC0
#define RIGHT_BOT 0xD9
int main(void)
{
   char i, j;
   /* draw the top of the box */
   putchar(LEFT_TOP);
   for (i=0; i<10; i++)
      putchar(HORIZ);
   putchar(RIGHT_TOP);
   putchar('\n');
   /* draw the middle */
   for (i=0; i<4; i++)
   {
      putchar(VERT);
      for (j=0; j<10; j++)
         putchar(' ');
      putchar(VERT);
      putchar('\n');
   }
   /* draw the bottom */
   putchar(LEFT_BOT);
   for (i=0; i<10; i++)
      putchar(HORIZ);
   putchar(RIGHT_BOT);
   putchar('\n');
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
putchar	+	+	+	+
putwchar		+	+	+

3.1.4.28.46 puts, _putws

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int puts(const char *s);
int _putws(const wchar_t *s);
```

Description

Outputs a string to stdout.

puts copies the null-terminated string s to the standard output stream stdout and appends a newline character.

Note: For Win32 GUI applications, stdout must be redirected.

Return Value

On successful completion, puts returns a nonnegative value. Otherwise, it returns a value of EOF.

Example

```
#include <stdio.h>
int main(void)
{
   char string[] = "This is an example output string\n";
   puts(string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
puts	+	+	+	+
_putws		+		

3.1.4.28.47 _putw

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _putw(int w, FILE *stream);
```

Description

Writes an integer on a stream.

_putw outputs the integer w to the given stream. _putw neither expects nor causes special alignment in the file.

Return Value

On success, _putw returns the integer w. On error, _putw returns EOF. Because EOF is a legitimate integer, use ferror to detect errors with _putw.

Example

```
#include <stdio.h>
#include <stdlib.h>
#define FNAME "test.$$$"
int main(void)
{
   FILE *fp;
   int word;
   /* place the word in a file */
   fp = fopen(FNAME, "wb");
   if (fp == NULL)
   {
      printf("Error opening file %s\n", FNAME);
      exit(1);
   }
   word = 94;
   putw(word,fp);
   if (ferror(fp))
       printf("Error writing to file\n");
   else
       printf("Successful write\n");
   fclose(fp);
   /* reopen the file */
   fp = fopen(FNAME, "rb");
   if (fp == NULL)
   {
      printf("Error opening file %s\n", FNAME);
      exit(1);
   }
   /* extract the word */
   word = getw(fp);
   if (ferror(fp))
       printf("Error reading file\n");
   else
       printf("Successful read: word = %d\n", word);
   /* clean up */
   fclose(fp);
   unlink(FNAME);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.28.48 remove, _wremove

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int remove(const char *filename);
```

```
int _wremove(const wchar_t *filename);
```

Description

Removes a file.

remove deletes the file specified by filename. It is a macro that simply translates its call to a call to unlink. If your file is open, be sure to close it before removing it.

The filename string can include a full path.

Return Value

On successful completion, remove returns 0. On error, it returns -1, and the global variable errno is set to one of the following values:

EACCES	Permission denied
ENOENT	No such file or directory

Example

```
#include <stdio.h>
int main(void)
{
    char file[80];
    /* prompt for file name to delete */
    printf("File to delete: ");
    gets(file);
    /* delete the file */
    if (remove(file) == 0)
       printf("Removed %s.\n",file);
    else
       perror("remove");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
remove	+	+	+	+
_wremove		NT only		

3.1.4.28.49 rename, _wrename

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int rename(const char *oldname, const char *newname);
```

```
int _wrename(const wchar_t *oldname, const wchar_t *newname);
```

Description

Renames a file.

rename changes the name of a file from oldname to newname. If a drive specifier is given in newname, the specifier must be the same as that given in oldname.

Directories in oldname and newname need not be the same, so rename can be used to move a file from one directory to another. Wildcards are not allowed.

This function will fail (EEXIST) if either file is currently open in any process.

Return Value

On success, rename returns 0.

On error (if the file cannot be renamed), it returns -1 and the global variable errno is set to one of the following values:

EEXIST	Permission denied: file already exists.
ENOENT	No such file or directory
ENOTSAM	Not same device

Example

```
#include <stdio.h>
int main(void)
{
    char oldname[80], newname[80];
    /* prompt for file to rename and new name */
    printf("File to rename: ");
    gets(oldname);
    printf("New name: ");
    gets(newname);
    /* Rename the file */
    if (rename(oldname, newname) == 0)
       printf("Renamed %s to %s.\n", oldname, newname);
    else
       perror("rename");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
rename	+	+	+	+
_wrename		NT only		

3.1.4.28.50 rewind

Header File

stdio.h

Category

Input/output Routines

Prototype

```
void rewind(FILE *stream);
```

Description

Repositions a file pointer to the beginning of a stream.

rewind(stream) is equivalent to fseek(stream, 0L, SEEK_SET), except that rewind clears the end-of-file and error indicators, while fseek clears the end-of-file indicator only.

After rewind, the next operation on an update file can be either input or output.

Return Value

None.

Example

```
#include <stdio.h>
#include <dir.h>
int main(void)
{
    FILE *fp;
    char *fname = "TXXXXXX", *newname, first;
    newname = mktemp(fname);
    fp = fopen(newname,"w+");
    fprintf(fp,"abcdefghijklmnopqrstuvwxyz");
    rewind(fp);
    fscanf(fp,"%c",&first);
    printf("The first character is: %c\n",first);
    fclose(fp);
    remove(newname);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.51 _rmtmp

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int _rmtmp(void);
```

Description

Removes temporary files.

The _rmtmp function closes and deletes all open temporary file streams which were previously created with tmpfile. The current directory must the same as when the files were created, or the files will not be deleted.

Return Value

_rmtmp returns the total number of temporary files it closed and deleted.

Example

```
#include <stdio.h>
#include <process.h>

void main()
{
   FILE *stream;
   int i;

   /* Create temporary files */
   for (i = 1; i <= 10; i++)
   {
     if ((stream = tmpfile()) == NULL)
       perror("Could not open temporary file\n");
     else
       printf("Temporary file %d created\n", i);
   }
   /* Remove temporary files */
   if (stream != NULL)
     printf("%d temporary files deleted\n", rmtmp());
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.28.52 scanf, wscanf

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int scanf(const char *format[, address, ...]);
```

```
int wscanf(const wchar_t *format[, address, ...]);
```

Description

Scans and formats input from the stdin stream.

Note: For Win32 GUI applications, stdin must be redirected.

The scanf function:

- scans a series of input fields one character at a time
- formats each field according to a corresponding format specifier passed in the format string *format.
- vsscanf scans and formats input from a string, using an argument list

There must be one format specifier and address for each input field.

scanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely. For details about why this might happen, see When ...scanf Stops Scanning.

Note: scanf often leads to unexpected results if you diverge from an expected pattern. You must provide information that tells scanf how to synchronize at the end of a line.

The combination of gets or fgets followed by sscanf is safe and easy, and therefore recommended over scanf.

Return Value

On success, scanf returns the number of input fields successfully scanned, converted, and stored. The return value does not include scanned fields that were not stored.

On error:

- if no fields were stored, scanf returns 0.
- if scanf attempts to read at end-of-file or at end-of-string, it returns EOF.

More About scanf

Example

```
#include <stdio.h>
int main(void)
{
   char label[20];
   char name[20];
   int entries = 0;
   int loop, age;
   double salary;
   struct Entry_struct
   {
      char  name[20];
      int   age;
      float salary;
   } entry[20];
/* Input a label as a string of characters restricting to 20 characters */
   printf("\n\nPlease enter a label for the chart: ");
   scanf("%20s", label);
   fflush(stdin);  /* flush the input stream in case of bad input */
/* Input number of entries as an integer */
  printf("How many entries will there be? (less than 20) ");
  scanf("%d", &entries);
  fflush(stdin);   /* flush the input stream in case of bad input */
/* input a name restricting input to only letters upper or lower case */
   for (loop=0;loop<entries;++loop)
   {
      printf("Entry %d\n", loop);
      printf("  Name   : ");
      scanf("%[A-Za-z]", entry[loop].name);
      fflush(stdin);  /* flush the input stream in case of bad input */
/* input an age as an integer */
      printf("  Age    : ");
      scanf("%d", &entry[loop].age);
      fflush(stdin);  /* flush the input stream in case of bad input */
/* input a salary as a float */
      printf("  Salary : ");
      scanf("%f", &entry[loop].salary);
      fflush(stdin); /* flush the input stream in case of bad input */
   }
/* Input a name, age and salary as a string, integer, and double */
   printf("\nPlease enter your name, age and salary\n");
   scanf("%20s %d %lf", name, &age, &salary);
/* Print out the data that was input */
   printf("\n\nTable %s\n",label);
   printf("Compiled by %s  age %d  $%15.2lf\n", name, age, salary);
   printf("-----------------------------------------------------\n");
   for (loop=0;loop<entries;++loop)
      printf("%4d | %-20s | %5d | %15.2lf\n",
         loop + 1,
         entry[loop].name,
         entry[loop].age,
         entry[loop].salary);
   printf("-----------------------------------------------------\n");
   return 0;
```

```
}
```

3.1.4.28.53 setbuf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
void setbuf(FILE *stream, char *buf);
```

Description

Assigns buffering to a stream.

setbuf causes the buffer buf to be used for I/O buffering instead of an automatically allocated buffer. It is used after stream has been opened.

If buf is null, I/O will be unbuffered; otherwise, it will be fully buffered. The buffer must be BUFSIZ bytes long (specified in stdio.h).

stdin and stdout are unbuffered if they are not redirected; otherwise, they are fully buffered. setbuf can be used to change the buffering style used.

Unbuffered means that characters written to a stream are immediately output to the file or device, while buffered means that the characters are accumulated and written as a block.

setbuf produces unpredictable results unless it is called immediately after opening stream or after a call to fseek. Calling setbuf after stream has been unbuffered is legal and will not cause problems.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file before returning from the function where the buffer was declared.

Return Value

None.

Example

```
#include <stdio.h>
/* BUFSIZ is defined in stdio.h */
char outbuf[BUFSIZ];
int main(void)
{
   /* attach a buffer to the standard output stream */
   setbuf(stdout, outbuf);
   /* put some characters into the buffer */
   puts("This is a test of buffered output.\n\n");
   puts("This output will go into outbuf\n");
   puts("and won't appear until the buffer\n");
   puts("fills up or we flush the stream.\n");
   /* flush the output buffer */
   fflush(stdout);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.54 setvbuf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Description

Assigns buffering to a stream.

setvbuf causes the buffer buf to be used for I/O buffering instead of an automatically allocated buffer. It is used after the given stream is opened.

If buf is null, a buffer will be allocated using malloc; the buffer will use size as the amount allocated. The buffer will be automatically freed on close. The size parameter specifies the buffer size and must be greater than zero.

The parameter size is limited by the constant UINT_MAX as defined in limits.h.

stdin and stdout are unbuffered if they are not redirected; otherwise, they are fully buffered. Unbuffered means that characters written to a stream are immediately output to the file or device, while buffered means that the characters are accumulated and written as a block.

- The type parameter is one of the following:

_IOFBF	fully buffered file. When a buffer is empty, the next input operation will attempt to fill the entire buffer. On output, the buffer will be completely filled before any data is written to the file.
_IOLBF	line buffered file. When a buffer is empty, the next input operation will still attempt to fill the entire buffer. On output, however, the buffer will be flushed whenever a newline character is written to the file.
_IONBF	unbuffered file. The buf and size parameters are ignored. Each input operation will read directly from the file, and each output operation will immediately write the data to the file.

A common cause for error is to allocate the buffer as an automatic (local) variable and then fail to close the file before returning from the function where the buffer was declared.

Return Value

On success, setvbuf returns 0.

On error (if an invalid value is given for type or size, or if there is not enough space to allocate a buffer), it returns nonzero.

Example

```
#include <stdio.h>
int main(void)
{
   FILE *input, *output;
   char bufr[512];
   input = fopen("file.in", "r+b");
   output = fopen("file.out", "w");
   /* set up input stream for minimal disk access,
      using our own character buffer */
```

```
if (setvbuf(input, bufr, _IOFBF, 512) != 0)
      printf("failed to set up buffer for input file\n");
   else
      printf("buffer set up for input file\n");
   /* set up output stream for line buffering using space that
      will be obtained through an indirect call to malloc */
   if (setvbuf(output, NULL, _IOLBF, 132) != 0)
      printf("failed to set up buffer for output file\n");
   else
      printf("buffer set up for output file\n");
   /* perform file I/O here */
   /* close files */
   fclose(input);
   fclose(output);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.55 snprintf;snwprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int snprintf(char* buffer, size_t nsize, const char* fmt, ...);
```

```
int snwprintf(wchar_t* buffer, size_t nsize, const wchar_t* fmt, ...);
```

Description

Sends formatted output to a buffer of a maximum length specified by nsize.

If the number of bytes to output is:

- < nsize, then all of the characters have been written, including the terminating '\0' character.
- == nsize, then nsize characters are written, with no terminating '\0' character.

> nsize, then only nsize characters are written, with no terminating '\0' character.

If nsize is 0, then the string will not be written to (and may be NULL).

Return Value

Number of bytes output, or, if nsize is 0, the number of bytes needed, not including the terminating '\0' character.

3.1.4.28.56 sprintf, swprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int sprintf(char *buffer, const char *format[, argument, ...]);
int swprintf(wchar_t *buffer, const wchar_t *format[, argument, ...]);
```

Description

Writes formatted output to a string.

Note: For details on format specifiers, see printf.

sprintf accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data to a string.

sprintf applies the first format specifier to the first argument, the second to the second, and so on. There must be the same number of format specifiers as arguments.

Return Value

On success, sprintf returns the number of bytes output. The return value does not include the terminating null byte in the count.

On error, sprintf returns EOF.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   char buffer[80];
   sprintf(buffer, "An approximation of Pi is %f\n", M_PI);
   puts(buffer);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sprintf	+	+	+	+
swprintf		+	+	+

3.1.4.28.57 sscanf, swscanf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Syntax

```
int sscanf(const char *buffer, const char *format[, address, ...]);
int swscanf(const wchar_t *buffer, const wchar_t *format[, address, ...]);
```

Description

Scans and formats input from a string.

Note: For details on format specifiers, see scanf.

sscanf scans a series of input fields, one character at a time, reading from a string. Then each field is formatted according to a format specifier passed to sscanf in the format string pointed to by format. Finally, sscanf stores the formatted input at an address passed to it as an argument following format. There must be the same number of format specifiers and addresses as there are input fields.

sscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See scanf for a discussion of possible causes.

Return Value

On success, sscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored.

If sscanf attempts to read at end-of-string, it returns EOF.

On error (If no fields were stored), it returns 0.

Example

```
#include <stdio.h>
#include <stdlib.h>
char *names[4] = {"Peter", "Mike", "Shea", "Jerry"};
#define NUMITEMS 4
int main(void)
{
   int   loop;
   char  temp[4][80];
   char  name[20];
   int   age;
   long  salary;
/* create name, age and salary data */
   for (loop=0; loop < NUMITEMS; ++loop)
      sprintf(temp[loop], "%s %d %ld", names[loop], random(10) + 20, random(5000) + 27500L);
/* print title bar */
   printf("%4s | %-20s | %5s | %15s\n", "#", "Name", "Age", "Salary");
   printf("   ------------------------------------------------\n");
/* input a name, age and salary data */
   for (loop=0; loop < NUMITEMS; ++loop)
      {
      sscanf(temp[loop],"%s %d %ld", &name, &age, &salary);
      printf("%4d | %-20s | %5d | %15ld\n", loop + 1, name, age, salary);
      }
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
sscanf	+	+	+	+
swscanf		+	+	+

3.1.4.28.58 stderr, stdin, stdout

Header File

stdio.h

Description

Predefined streams automatically opened when the program is started.

Name	Meaning
stdin	Standard input device
stdout	Standard output device
stderr	Standard error output device

3.1.4.28.59 _tempnam, _wtempnam

Header File

stdio.h

Category

Input/output Routines

Prototype

```
char *_tempnam(char *dir, char *prefix)
```

```
wchar_t *_wtempnam(wchar_t *dir, wchar_t *prefix)
```

Description

Creates a unique file name in specified directory.

The _tempnam function accepts single-byte or multibyte string arguments.

The _tempnam function creates a unique file name in arbitrary directories. The unique file is not actually created; _tempnam only verifies that it does not currently exist. It attempts to use the following directories, in the order shown, when creating the file name:

- The directory specified by the TMP environment variable.
- The dir argument to _tempnam.
- The P_tmpdir definition in stdio.h. If you edit stdio.h and change this definition, _tempnam will not use the new definition.
- The current working directory.

If any of these directories is NULL, or undefined, or does not exist, it is skipped.

The prefix argument specifies the first part of the file name; it cannot be longer than 5 characters, and cannot contain a period (.). A unique file name is created by concatenating the directory name, the prefix, and 6 unique characters. Space for the resulting file name is allocated with malloc; when this file name is no longer needed, the caller should call free to free it.

If you do create a temporary file using the name constructed by _tempnam, it is your responsibility to delete the file name (for example, with a call to remove). It is not deleted automatically. (tmpfile does delete the file name.)

Return Value

If _tempnam is successful, it returns a pointer to the unique temporary file name, which the caller can pass to free when it is no longer needed. Otherwise, if _tempnam cannot create a unique file name, it returns NULL.

Example

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  FILE *stream;
  int i;
  char *name;
  for (i = 1; i <= 10; i++) {
    if ((name = tempnam("\\tmp","wow")) == NULL)
      perror("tempnam couldn't create name");
```

```
    else {
      printf("Creating %s\n",name);
      if ((stream = fopen(name,"wb")) == NULL)
        perror("Could not open temporary file\n");
      else
        fclose(stream);
    }
    free(name);
  }
  printf("Warning: temp files not deleted.\n");
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_tempnam		+		
_wtempnam		+		

3.1.4.28.60 tmpfile

Header File

stdio.h

Category

Input/output Routines

Prototype

```
FILE *tmpfile(void);
```

Description

Opens a “scratch” file in binary mode.

tmpfile creates a temporary binary file and opens it for update (w + b). If you do not change the directory after creating the temporary file, the file is automatically removed when it’s closed or when your program terminates.

Return Value

tmpfile returns a pointer to the stream of the temporary file created. If the file can’t be created, tmpfile returns NULL.

Example

```
#include <stdio.h>
#include <process.h>
int main(void)
{
   FILE *tempfp;
   tempfp = tmpfile();
   if (tempfp)
      printf("Temporary file created\n");
   else
   {
      printf("Unable to create temporary file\n");
      exit(1);
   }
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.28.61 tmpnam, _wtmpnam

Header File

stdio.h

Category

Input/output Routines

Prototype

```
char *tmpnam(char *s);
```

```
wchar_t *_wtmpnam(wchar_t *s);
```

Description

Creates a unique file name.

tmpnam creates a unique file name, which can safely be used as the name of a temporary file. tmpnam generates a different string each time you call it, up to TMP_MAX times. TMP_MAX is defined in stdio.h as 65,535.

The parameter to tmpnam, s, is either null or a pointer to an array of at least L_tmpnam characters. L_tmpnam is defined in stdio.h. If s is NULL, tmpnam leaves the generated temporary file name in an internal static object and returns a pointer to that object. If s is not NULL, tmpnam overwrites the internal static object and places its result in the pointed-to array, which must be at least L_tmpnam characters long, and returns s.

If you do create such a temporary file with tmpnam, it is your responsibility to delete the file name (for example, with a call to remove). It is not deleted automatically. (tmpfile does delete the file name.)

Return Value

If s is null, tmpnam returns a pointer to an internal static object. Otherwise, tmpnam returns s.

Example

```
#include <stdio.h>
int main(void)
{
   char name[13];
   tmpnam(name);
   printf("Temporary name: %s\n", name);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
tmpnam	+	+	+	+
_wtmpnam		+		

3.1.4.28.62 ungetc, ungetwc

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int ungetc(int c, FILE *stream);
```

```
wint_t ungetwc(wint_t c, FILE *stream);
```

Description

Pushes a character back into input stream.

Note: Do not use this function in Win32 GUI applications.

ungetc pushes the character c back onto the named input stream, which must be open for reading. This character will be returned on the next call to getc or fread for that stream. One character can be pushed back in all situations. A second call to ungetc without a call to getc will force the previous character to be forgotten. A call to fflush, fseek, fsetpos, or rewind erases all memory of any pushed-back characters.

Return Value

On success, ungetc returns the character pushed back.

On error, it returns EOF.

Example

```
#include <stdio.h>
#include <ctype.h>
int main( void )
{
   int i=0;
   char ch;
   puts("Input an integer followed by a char:");
   /* read chars until non digit or EOF */
   while((ch = getchar()) != EOF && isdigit(ch))
      i = 10 * i + ch - 48; /* convert ASCII into int value */
   /* if non digit char was read, push it back into input buffer */
   if (ch != EOF)
      ungetc(ch, stdin);
   printf("i = %d, next char in buffer = %c\n", i, getchar());
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ungetc	+	+	+	+
ungetwc		+	+	+

3.1.4.28.63 vfprintf, vfwprintf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int vfprintf(FILE *stream, const char *format, va_list arglist);

int vfwprintf(FILE *stream, const wchar_t *format, va_list arglist);
```

Description

Writes formatted output to a stream.

The v...printf functions are known as alternate entry points for the ...printf functions. They behave exactly like their ...printf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Printf Format Specifiers.

vfprintf accepts a pointer to a series of arguments, applies to each argument a format specifier contained in the format string pointed to by format, and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.

Return Value

On success, vfprintf returns the number of bytes output.

On error, it returns EOF.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
FILE *fp;
int vfpf(char *fmt, ...)
{
   va_list argptr;
   int cnt;
   va_start(argptr, fmt);
   cnt = vfprintf(fp, fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
   char string[4] = "abc";
   fp = tmpfile();
   if (fp == NULL)
   {
      perror("tmpfile() call");
      exit(1);
   }
   vfpf("%d %f %s", inumber, fnumber, string);
   rewind(fp);
   fscanf(fp,"%d %f %s", &inumber, &fnumber, string);
   printf("%d %f %s\n", inumber, fnumber, string);
   fclose(fp);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
vfprintf	+	+	+	+
vfwprintf		+	+	+

3.1.4.28.64 vfscanf

Header File

stdio.h

Category

Input/output Routines

Prototype

```
int vfscanf(FILE *stream, const char *format,va_list arglist);
```

Description

Scans and formats input from a stream.

The v...scanf functions are known as alternate entry points for the ...scanf functions. They behave exactly like their ...scanf counterparts but they accept a pointer to a list of arguments instead of an argument list.

For details on format specifiers, see Scanf Format Specifiers.

vfscanf scans a series of input fields one character at a time reading from a stream. Then each field is formatted according to a format specifier passed to vfscanf in the format string pointed to by format. Finally vfscanf stores the formatted input at an address passed to it as an argument following format. There must be the same number of format specifiers and addresses as there are input fields. vfscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character or it might terminate entirely for a number of reasons. See scanf for a discussion of possible causes.

Return Value

vfscanf returns the number of input fields successfully scanned converted and stored; the return value does not include scanned fields that were not stored. If no fields were stored the return value is 0.

If vfscanf attempts to read at end-of-file the return value is EOF.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
FILE *fp;
int vfsf(char *fmt, ...)
{
   va_list  argptr;
   int cnt;
   va_start(argptr, fmt);
   cnt = vfscanf(fp, fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
         char string[4] = "abc";
   fp = tmpfile();
   if (fp == NULL)
   {
      perror("tmpfile() call");
      exit(1);
   }
   fprintf(fp,"%d %f %s\n",inumber,fnumber,string);
   rewind(fp);
```

```
   vfsf("%d %f %s",&inumber,&fnumber,string);
   printf("%d %f %s\n",inumber,fnumber,string);
   fclose(fp);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+	+	+

3.1.4.28.65 vprintf, vwprintf

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int vprintf(const char *format, va_list arglist);
```

```
int vwprintf(const wchar_t * format, va_list arglist);
```

Description

Writes formatted output to stdout.

Note: Do not use this function in Win32 GUI applications.

The v...printf functions are known as alternate entry points for the ...printf functions. They behave exactly like their ...printf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

Note: For details on format specifiers, see Printf Format Specifiers.

vprintf accepts a pointer to a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data to stdout. There must be the same number of format specifiers as arguments.

Return Value

vprint returns the number of bytes output. In the event of error, vprint returns EOF.

Example

```
#include <stdio.h>
#include <stdarg.h>
int vpf(char *fmt, ...)
{
   va_list argptr;
   int cnt;
   va_start(argptr, fmt);
   cnt = vprintf(fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
   char *string = "abc";
   vpf("%d %f %s\n",inumber,fnumber,string);
```

```
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
vprintf	+	+	+	+
vwprintf		+	+	+

3.1.4.28.66 vscanf

Header File

stdio.h

Category

Console I/O Routines

Prototype

```
int vscanf(const char *format, va_list arglist);
```

Description

Scans and formats input from stdin.

Note: Do not use this function in Win32 GUI applications.

The v...scanf functions are known as alternate entry points for the ...scanf functions. They behave exactly like their ...scanf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

Note: For details on format specifiers, see Scanf Format Specifiers.

vscanf scans a series of input fields, one character at a time, reading from stdin. Then each field is formatted according to a format specifier passed to vscanf in the format string pointed to by format. Finally, vscanf stores the formatted input at an address passed to it as an argument following format. There must be the same number of format specifiers and addresses as there are input fields.

vscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See scanf for a discussion of possible causes.

Return Value

vscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If vscanf attempts to read at end-of-file, the return value is EOF.

Example

```
#include <stdio.h>
#include <stdarg.h>
int vscnf(char *fmt, ...)
{
   va_list argptr;
   int cnt;
   printf("Enter an integer, a float, and a string (e.g. i,f,s,)\n");
   va_start(argptr, fmt);
   cnt = vscanf(fmt, argptr);
   va_end(argptr);
   return(cnt);
```

```
}
int main(void)
{
   int inumber;
   float fnumber;
   char string[80];
   vscnf("%d, %f, %s", &inumber, &fnumber, string);
   printf("%d %f %s\n", inumber, fnumber, string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+	+	+

3.1.4.28.67 vsnprintf;vsnwprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int vsnprintf(char* buffer, size_t nsize, const char* format, va_list param);
```

```
int vsnwprintf(wchar_t* buffer, size_t nsize, const wchar_t* format, va_list param);
```

Description

Sends formatted output to a buffer of maximum length specified by nsize.

If the number of bytes to output is:

- < nsize, then all of the characters have been written, including the terminating ‘\0’ character.
- == nsize, then nsize characters are written, with no terminating ‘\0’ character.

> nsize, then only nsize characters are written, with no terminating ‘\0’ character.

If nsize is 0, then the string will not be written to (and may be NULL).

Return Value

Number of bytes output, or, if nsize is 0, the number of bytes needed, not including the terminating ‘\0’ character.

3.1.4.28.68 vsprintf, vswprintf

Header File

stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
int vsprintf(char *buffer, const char *format, va_list arglist);
```

```
int vswprintf(wchar_t *buffer, const wchar_t *format, va_list arglist);
```

Description

Writes formatted output to a string.

The v...printf functions are known as alternate entry points for the ...printf functions. They behave exactly like their ...printf counterparts, but they accept a pointer to a list of arguments instead of an argument list.

vsprintf accepts a pointer to a series of arguments, applies to each a format specifier contained in the format string pointed to by format, and outputs the formatted data to a string. There must be the same number of format specifiers as arguments.

Return Value

vsprintf returns the number of bytes output. In the event of error, vsprintf returns EOF.

Example

```
#include <stdio.h>
#include <conio.h>
#include <stdarg.h>
char buffer[80];
int vspf(char *fmt, ...)
{
   va_list argptr;
   int cnt;
   va_start(argptr, fmt);
   cnt = vsprintf(buffer, fmt, argptr);
   va_end(argptr);
   return(cnt);
}
int main(void)
{
   int inumber = 30;
   float fnumber = 90.0;
   char string[4] = "abc";
   vspf("%d %f %s", inumber, fnumber, string);
   printf("%s\n", buffer);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
vsprintf	+	+	+	+
vswprintf		+	+	+

3.1.4.29 stdlib.h

The following functions, macros, and classes are provided in `stdlib.h`:

Topics

Name	Description
EXIT_xxxx #defines (see page 1095)	**Header File** stdlib.h **Description** Constants defining exit conditions for calls to the exit function.
RAND_MAX #define (see page 1095)	**Header File** stdlib.h **Syntax** **Description** Maximum value returned by rand function.

_argc (see page 1095)	**Header File** stdlib.h **Syntax** `extern int _argc;` **Description** _argc has the same value as argc (passed to main) when the program starts. This variable holds the number of arguments passed to the program. The value includes the name of the program itself, so _argc and argc are always at least 1.

_argv, _wargv (see page 1095)	**Header File**
	stdlib.h
	Syntax
	`extern char **_argv;`
	`extern wchar_t ** _wargv`
	Description
	_argv points to an array containing the original command-line arguments (the elements of argv[]) passed to main when the program starts.
	_wargv is the Unicode version of _argv.
	Portability

_atoi64, _wtoi64 (see page 1096)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`__int64 _atoi64(const char *s);`
	`__int64 _wtoi64(const wchar_t *s);`
	Description
	Converts a string to an **__int64**.
	The syntax of the string must be:
	`__int64 ::= [isspace]* [sign] digit [digit]*`
	Only decimal integers are acceptable.
	_wtoi64 is the wide-character version. It converts a wide-character string to an **__int64**.
	In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in atoi (results are undefined). There is no defined method to return an error indication to the caller. The result is undefined if the input string... more (see page 1096)

_crotl, _crotr (see page 1096)	**Header File**
	stdlib.h
	Category
	Math Routines
	Prototype
	`unsigned char _crotl(unsigned char val, int count);`
	`unsigned char _crotr(unsigned char val, int count);`
	Description
	Rotates an unsigned char left or right.
	_crotl rotates the given val to the left count bits. _crotr rotates the given val to the right count bits.
	The argument val is an **unsigned char**, or its equivalent in decimal or hexadecimal form.
	Return Value
	• The functions return the rotated byte:
	• _crotl returns the value of val left-rotated count bits.
	• _crotr returns the value of val right-rotated count bits.
	Portability

_exit (see page 1097)	**Header File** stdlib.h **Category** Process Control Routines **Prototype** `void _exit(int status);` **Description** Terminates program. _exit terminates execution without closing any files, flushing any output, or calling any exit functions. The calling process uses status as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error. **Return Value** None. **Example**

_fullpath, _wfullpath (see page 1098)	**Header File**
	stdlib.h
	Category
	Directory Control Routines
	Prototype
	`char * _fullpath(char *buffer, const char *path, int buflen);`
	`wchar_t * _wfullpath(wchar_t *buffer, const wchar_t *path, int buflen);`
	Description
	Converts a path name from relative to absolute.
	_fullpath converts the relative path name in path to an absolute path name that is stored in the array of characters pointed to by buffer. The maximum number of characters that can be stored at buffer is buflen. The function returns NULL if the buffer isn't big enough to store the absolute path name or if the path contains an invalid drive letter.
	If buffer... more (see page 1098)

_lrand (see page 1099)	**Header File** stdlib.h **Category** Math Routines **Prototype** `long _lrand(void);` **Description** _lrand is the long random number generator function. _rand uses a multiplicative congruential random number generator with period 2^64 to return successive pseudo-random numbers in the range from 0 to 2^31 - 1. The generator is reinitialized by calling srand with an argument value of 1. It can be set to a new starting point by calling srand with a given seed number.

_lrotl, _lrotr (see page 1099)	**Header File**
	stdlib.h
	Category
	Math Routines
	Prototype
	`unsigned long _lrotl(unsigned long val, int count);`
	`unsigned long _lrotr(unsigned long val, int count);`
	Description
	Rotates an **unsigned long** integer value to the left or right.
	_lrotlrotates the given val to the left count bits. _lrotr rotates the given val to the right count bits.
	Return Value
	The functions return the rotated integer:
	• _lrotl returns the value of val left-rotated count bits.
	• _lrotr returns the value of val right-rotated count bits.
	Example

_makepath, _wmakepath (see page 1100)	**Header File**
	stdlib.h
	Category
	Directory Control Routines
	Prototype
	`void _makepath(char *path, const char *drive, const char *dir, const char *name, const char *ext);`
	`void _wmakepath(wchar_t *path, const wchar_t *drive, const wchar_t *dir, const wchar_t *name, const wchar_t *ext);`
	Description
	Builds a path from component parts.
	_makepath makes a path name from its components. The new path name is
	`X:\DIR\SUBDIR\NAME.EXT`
	where

_rotl, _rotr (see page 1102)	**Header File**
	stdlib.h
	Category
	Math Routines
	Prototype
	`unsigned short _rotl(unsigned short value, int count);`
	`unsigned short _rotr(unsigned short value, int count);`
	Description
	Bit-rotates an **unsigned** short integer value to the left or right.
	_rotl rotates the given value to the left count bits.
	_rotr rotates the given value to the right count bits.
	Return Value
	_rotl, and _rotr return the rotated integer:
	• _rotl returns the value of value left-rotated count bits.
	• _rotr returns the value of value right-rotated count bits.
	Example

_searchenv, _wsearchenv (see page 1103)	**Header File**
	stdlib.h
	Category
	Miscellaneous Routines
	Prototype
	`char *_searchenv(const char *file, const char *varname, char *buf);`
	`char *_wsearchenv(const wchar_t *file, const wchar_t *varname, wchar_t *buf);`
	Description
	Looks for a file, using an environment variable as the search path.
	_searchenv attempts to locate file, searching along the path specified by the operating system environment variable varname. Typical environment variables that contain paths are PATH, LIB, and INCLUDE.
	_searchenv searches for the file in the current directory of the current drive first. If the file is not found there, the environment variable varname is fetched, and each directory in the path it... more (see page 1103)

_searchstr, _wsearchstr (see page 1104)	**Header File**
	stdlib.h
	Category
	Miscellaneous Routines
	Prototype
	`void _searchstr(const char *file, const char *ipath, char *buf);`
	`void _wsearchstr(const wchar_t *file, const wchar_t *ipath,wchar_t *pathname);`
	Description
	Searches a list of directories for a file.
	_searchstr attempts to locate file, searching along the path specified by the string ipath.
	_searchstr searches for the file in the current directory of the current drive first. If the file is not found there, each directory in ipath is searched in turn until the file is found, or the path is exhausted. The directories in ipath must be separated by semicolons.
	When the file is located,... more (see page 1104)

_splitpath, _wsplitpath (see page 1105)	**Header File**
	stdlib.h
	Category
	Directory Control Routines
	Prototype
	`void _splitpath(const char *path, char *drive, char *dir, char *name, char *ext);`
	`void _wsplitpath(const wchar_t *path, wchar_t *drive, wchar_t *dir, wchar_t *name, wchar_t *ext);`
	Description
	Splits a full path name into its components.
	_splitpath takes a file's full path name (path) as a string in the form
	X:\DIR\SUBDIR\NAME.EXT
	and splits path into its four components. It then stores those components in the strings pointed to by drive, dir, name, and ext. (All five components must be passed, but any of them can be a null, which means the corresponding component will be... more (see page 1105)

_ui64toa, _ui64tow (see page 1106)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`char *_ui64toa(unsigned __int64 value, char *string, int radix);`
	`wchar_t *_ui64tow(unsigned __int64 value, wchar_t *string, int radix);`
	Description
	_ui64toa converts an unsigned **__int64** to a string.
	_ui64tow is the unicode version. _ui64tow converts an unsigned **__int64** to a wide-character string.
	These functions convert value to a null-terminated string and store the result in string. value is an **__int64**.
	radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is... more (see page 1106)

abort (see page 1107)	**Header File** stdlib.h **Category** Process Control Routines **Prototype** `void abort(void);` **Description** Abnormally terminates a program. abort causes an abnormal program termination by calling raise(SIGABRT). If there is no signal handler for SIGABRT, then abort writes a termination message (Abnormal program termination) on stderr, then aborts the program by a call to _exit with exit code 3. **Return Value** abort returns the exit code 3 to the parent process or to the operating system command processor. **Example**
atexit (see page 1107)	**Header File** stdlib.h **Category** Process Control Routines **Prototype** `int atexit(void (_USERENTRY * func)(void));` **Description** Registers termination function. atexit registers the function pointed to by func as an exit function. Upon normal termination of the program, exit calls func just before returning to the operating system. fcmp must be used with the _USERENTRY calling convention. Each call to atexit registers another exit function. Up to 32 functions can be registered. They are executed on a last-in, first-out basis (that is, the last function registered is the first to be executed). **Return Value** atexit returns 0 on success and nonzero on failure... more (see page 1107)

atoi, _wtoi (see page 1108)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`int atoi(const char *s);`
	`int _wtoi(const wchar_t *s);`
	Description
	Converts a string to an integer.
	• atoi converts a string pointed to by s to **int**; atoi recognizes (in the following order)
	• An optional string of tabs and spaces
	• An optional sign
	• A string of digits
	The characters must match this generic format:
	[ws] [sn] [ddd]
	In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in atoi (results are undefined).
	Return Value
	atoi returns the converted value of the input string. If the... more (see page 1108)

atol, _wtol (see page 1109)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`long atol(const char *s);`
	`long _wtol(const wchar_t *s);`
	Description
	Converts a string to a long.
	• atol converts the string pointed to by s to **long**. atol recognizes (in the following order)
	• An optional string of tabs and spaces
	• An optional sign
	• A string of digits
	The characters must match this generic format:
	`[ws] [sn] [ddd]`
	In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in atol (results are undefined).
	Return Value
	atol returns the converted value of the input string. If the... more (see page 1109)

bsearch (see page 1110)	**Header File** stdlib.h **Category** Memory and String Manipulation Routines **Prototype** `void *bsearch(const void *key, const void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));` **Description** Binary search of an array. bsearch searches a table (array) of nelem elements in memory, and returns the address of the first entry in the table that matches the search key. The array must be in order. If no match is found, bsearch returns 0. **Note:** Because this is a binary search, the first matching entry is not necessarily the first entry in the table. The type size_t is defined... more (see page 1110)

div (see page 1111)	**Header File**
	stdlib.h
	Category
	Math Routines
	Prototype
	`div_t div(int numer, int denom);`
	Description
	Divides two integers, returning quotient and remainder.
	div divides two integers and returns both the quotient and the remainder as a div_t type. numer and denom are the numerator and denominator, respectively. The div_t type is a structure of integers defined (with **typedef**) in stdlib.h as follows:
	`typedef struct {`
	`int quot; /* quotient */`
	`int rem; /* remainder */`
	`} div_t;`
	Return Value
	div returns a structure whose elements are quot (the quotient) and rem (the remainder).
	Example

ecvt (see page 1112)	**Header File** stdlib.h **Category** Conversion Routines, Math Routines **Prototype** `char *ecvt(double value, int ndig, int *dec, int *sign);` **Description** Converts a floating-point number to a string. ecvt converts value to a null-terminated string of ndig digits, starting with the leftmost significant digit, and returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through dec (a negative value for dec means that the decimal lies to the left of the returned digits). There is no decimal point in the string itself. If the sign of value is... more (see page 1112)

_environ, _wenviron (see page 1113)	**Header File**
	stdlib.h
	Syntax
	`extern char ** _environ;`
	`extern wchar_t ** _wenviron`
	Description
	_environ is an array of pointers to strings; it is used to access and alter the operating system environment variables. Each string is of the form:
	`envvar = varvalue`
	where envvar is the name of an environment variable (such as PATH), and varvalue is the string value to which envvar is set (such as C:\Utils;C:\MyPrograms). The string varvalue can be empty.
	When a program begins execution, the operating system environment settings are passed directly to the program. Note that env, the third argument to main, is equal... more (see page 1113)

exit (see page 1114)	**Header File** stdlib.h **Category** Process Control Routines **Prototype** `void exit(int status);` **Description** Terminates program. exit terminates the calling process. Before termination, all files are closed, buffered output (waiting to be output) is written, and any registered "exit functions" (posted with atexit) are called. status is provided for the calling process as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error. It can be, but is not required, to be set with one of the following:
fcvt (see page 1114)	**Header File** stdlib.h **Category** Conversion Routines, Math Routines **Prototype** `char *fcvt(double value, int ndig, int *dec, int *sign);` **Description** Converts a floating-point number to a string. fcvt converts value to a null-terminated string digit starting with the leftmost significant digit with ndig digits to the right of the decimal point. fcvt then returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through dec (a negative value for dec means to the left of the returned digits). There is no decimal point in the string itself. If... more (see page 1114)

gcvt (see page 1115)	**Header File** stdlib.h **Category** Conversion Routines, Math Routines **Prototype** `char *gcvt(double value, int ndec, char *buf);` **Description** Converts floating-point number to a string. gcvt converts value to a null-terminated ASCII string and stores the string in buf. It produces ndec significant digits in FORTRAN F format, if possible; otherwise, it returns the value in the printf E format (ready for printing). It might suppress trailing zeros. **Return Value** gcvt returns the address of the string pointed to by buf. **Example**

getenv, _wgetenv (see page 1116)	**Header File**
	stdlib.h
	Category
	Process Control Routines
	Prototype
	`char *getenv(const char *name);`
	`wchar_t *_wgetenv(const wchar_t *name);`
	Description
	Find or delete an environment variable from the system environment.
	The environment consists of a series of entries that are of the form name=string\0.
	getenv returns the value of a specified variable. name can be either uppercase or lowercase. name must not include the equal sign (**=**). If the specified environment variable does not exist, getenv returns a NULL pointer.
	To delete the variable from the environment, use getenv("name=").
	Note: Environment entries must not be changed directly. If you want to... more (see page 1116)

itoa, _itow (see page 1117)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`char *itoa(int value, char *string, int radix);`
	`wchar_t *_itow(int value, wchar_t *string, int radix);`
	Description
	Converts an integer to a string.
	itoa converts value to a null-terminated string and stores the result in string. With itoa, value is an integer. _itow is the unicode version of the function. It converts an integer to a wide-character string.
	radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus... more (see page 1117)

labs (see page 1118)	**Header File** stdlib.h **Category** Math Routines **Prototype** `long labs(long int x);` **Description** Gives long absolute value. labs computes the absolute value of the parameter x. **Return Value** labs returns the absolute value of x. **Example**

lfind (see page 1119)	**Header File** stdlib.h **Category** Memory and String Manipulation Routines **Prototype** `void *lfind(const void *key, const void *base, size_t *num, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));` **Description** Performs a linear search. lfind makes a linear search for the value of key in an array of sequential records. It uses a user-defined comparison routine fcmp. The fcmp function must be used with the _USERENTRY calling convention. The array is described as having *num records that are width bytes wide, and begins at the memory location pointed to by base. **Return Value** lfind returns the address of the first... more (see page 1119)
lsearch (see page 1120)	**Header File** stdlib.h **Category** Memory and String Manipulation Routines **Prototype** `void *lsearch(const void *key, void *base, size_t *num, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));` **Description** Performs a linear search. lsearch searches a table for information. Because this is a linear search, the table entries do not need to be sorted before a call to lsearch. If the item that key points to is not in the table, lsearch appends that item to the table. • base points to the base (0th element) of the search table. • num points to an integer containing the number of entries in... more (see page 1120)

ltoa, _ltoa, _ltow (see page 1121)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`char * ltoa(long value, char * string, int radix);`
	`char *_ltoa(long value, char *string, int radix);`
	`wchar_t *_ltow(long value, wchar_t *string, int radix);`
	Description
	Converts a **long** to a string. _ltow is the unicode version. It converts a **long** to a wide-charater string.
	Converts value to a null-terminated string and stores the result in string. value is a **long**.
	radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of... more (see page 1121)

max (see page 1122)	**Header File**
	stdlib.h
	Category
	C++ Prototyped Routines
	Prototype
	`(type) max(a, b);`
	`template <class T> T max( T t1, T t2 ); // C++ only`
	Description
	Returns the larger of two values.
	The C macro and the C++ template function compare two values and return the larger of the two. Both arguments and the routine declaration must be of the same type.
	Return Value
	max returns the larger of two values.
	Portability

mblen (see page 1122)	**Header File** stdlib.h **Category** Memory and String Manipulation Routines **Prototype** `int mblen(const char *s, size_t n);` **Description** Determines the length of a multibyte character. If s is not null, mblen determines the number of bytes in the multibyte character pointed to by s. The maximum number of bytes examined is specified by n. The behavior of mblen is affected by the setting of LC_CTYPE category of the current locale. **Return Value** If s is null, mblen returns a nonzero value if multibyte characters have state-dependent encodings. Otherwise, mblen returns 0. If s is not null, mblen returns 0 if s... more (see page 1122)
mbstowcs (see page 1123)	**Header File** stdlib.h **Category** Conversion Routines, Memory and String Manipulation Routines **Prototype** `size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);` **Description** Converts a multibyte string to a **wchar_t** array. The function converts the multibyte string s into the array pointed to by pwcs. No more than n values are stored in the array. If an invalid multibyte sequence is encountered, mbstowcs returns (size_t) -1. The pwcs array will not be terminated with a zero value if mbstowcs returns n. **Return Value** If an invalid multibyte sequence is encountered, mbstowcs returns (size_t) -1. Otherwise, the function returns the number of array... more (see page 1123)
mbtowc (see page 1124)	**Header File** stdlib.h **Category** Conversion Routines, Memory and String Manipulation Routines **Prototype** `int mbtowc(wchar_t *pwc, const char *s, size_t n);` **Description** Converts a multibyte character to **wchar_t** code. If s is not null, mbtowc determines the number of bytes that comprise the multibyte character pointed to by s. Next, mbtowc determines the value of the type **wchar_t** that corresponds to that multibyte character. If there is a successful match between **wchar_t** and the multibyte character, and pwc is not null, the **wchar_t** value is stored in the array pointed to by pwc. At most n characters are examined. **Return Value... more (see page 1124)**

min (see page 1125)	**Header File**
	stdlib.h
	Category
	C++ Prototyped Routines
	Prototype
	`(type) min(a, b); /* macro version */`
	`template <class T> T min( T t1, T t2 );// C++ only`
	Description
	Returns the smaller of two values.
	The C macro and the C++ template function compare two values and return the smaller of the two. Both arguments and the routine declaration must be of the same type.
	Return Value
	min returns the smaller of two values.

putenv, _wputenv (see page 1126)	**Header File**
	stdlib.h
	Category
	Process Control Routines
	Prototype
	`int putenv(const char *name);`
	`int _wputenv(const wchar_t *name);`
	Description
	Adds string to current environment.
	putenv accepts the string name and adds it to the environment of the current process. For example,
	`putenv("PATH=C:\\BC");`
	putenv can also be used to modify an existing name. name can be either uppercase or lowercase. name must not include the equal sign (=). You can set a variable to an empty value by specifying an empty string on the right side of the '=' sign.
	putenv can be used only to modify the current program's _environment.... more (see page 1126)

qsort (see page 1127)	**Header File** stdlib.h **Category** Memory and String Manipulation Routines **Prototype** `void qsort(void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));` **Description** Sorts using the quicksort algorithm. qsort is an implementation of the "median of three" variant of the quicksort algorithm. qsort sorts the entries in a table by repeatedly calling the user-defined comparison function pointed to by fcmp. • base points to the base (0th element) of the table to be sorted. • nelem is the number of entries in the table. • width is the size of each entry in the table, in bytes. fcmp, the comparison... more (see page 1127)
rand (see page 1128)	**Header File** stdlib.h **Category** Math Routines **Prototype** `int rand(void);` **Description** Random number generator. rand uses a multiplicative congruential random number generator with period 2 to the 32nd power to return successive pseudo-random numbers in the range from 0 to RAND_MAX. The symbolic constant RAND_MAX is defined in stdlib.h. **Return Value** rand returns the generated pseudo-random number. **Example**

random (see page 1128)	**Header File** stdlib.h **Category** Math Routines **Prototype** `int random(int num);` **Description** Random number generator. random returns a random number between 0 and (num-1). random(num) is a macro defined in stdlib.h. Both num and the random number returned are integers. **Return Value** random returns a number between 0 and (num-1). **Example**
randomize (see page 1129)	**Header File** stdlib.h, time.h **Category** Math Routines **Prototype** `void randomize(void);` **Description** Initializes random number generator. randomize initializes the random number generator with a random value. **Return Value** None. **Example**
srand (see page 1130)	**Header File** stdlib.h **Category** Math Routines **Prototype** `void srand(unsigned seed);` **Description** Initializes random number generator. The random number generator is reinitialized by calling srand with an argument value of 1. It can be set to a new starting point by calling srand with a given seed number. **Return Value** None. **Example**

strlen, _mbslen, wcslen, _mbstrlen (see page 1131)	**Header File**
	string.h, mbstring.h, stdlib.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`size_t strlen(const char *s);`
	`size_t wcslen(const wchar_t *s);`
	`size_t _mbslen(const unsigned char *s);`
	`size_t _mbstrlen(const char *s)`
	Description
	Calculates the length of a string.
	strlen calculates the length of s.
	_mbslen and _mbstrlen test the string argument to determine the number of multibyte characters they contain.
	_mbstrlen is affected by the LC_CTYPE category setting as determined by the setlocale function. The function tests to determine whether the string argument is a valid multibyte string.
	_mbslen is affected by the code page that is in use. This... more (see page 1131)

strtod, _strtold, wcstod, _wcstold (see page 1131)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`double strtod(const char *s, char **endptr);`
	`double wcstod(const wchar_t *s, wchar_t **endptr);`
	`long double _strtold(const char *s, char **endptr);`
	`long double _wcstold(const wchar_t *s, wchar_t **endptr);`
	Description
	Convert a string to a **double** or **long double** value.
	strtod converts a character string, s, to a **double** value. s is a sequence of characters that can be interpreted as a **double** value; the characters must match this generic format:
	`[ws] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]`
	where:

strtol, wcstol (see page 1133)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`long strtol(const char *s, char **endptr, int radix);`
	`long wcstol(const wchar_t *s, wchar_t **endptr, int radix);`
	Description
	Converts a string to a **long** value.
	strtol converts a character string, s, to a **long** integer value. s is a sequence of characters that can be interpreted as a **long** value; the characters must match this generic format:
	`[ws] [sn] [0] [x] [ddd]`
	where:

strtoul, wcstoul (see page 1134)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`unsigned long strtoul(const char *s, char **endptr, int radix);`
	`unsigned long wcstoul(const wchar_t *s, wchar_t **endptr, int radix);`
	Description
	Converts a string to an **unsigned long** in the given radix.
	strtoul operates the same as strtol, except that it converts a string str to an **unsigned long** value (where strtol converts to a **long**). Refer to the entry for strtol for more information.
	Return Value
	strtoul returns the converted value, an **unsigned long**, or 0 on error.
	Example

swab (see page 1135)	**Header File** stdlib.h **Category** Memory and String Manipulation Routines **Prototype** `void swab(char *from, char *to, int nbytes);` **Description** Swaps bytes. swab copies nbytes bytes from the from string to the to string. Adjacent even- and odd-byte positions are swapped. This is useful for moving data from one machine to another machine with a different byte order. nbytes should be even. **Return Value** None. **Example**

system, _wsystem (see page 1136)	**Header File**
	stdlib.h
	Category
	Process Control Routines
	Prototype
	`int system(const char *command);`
	`int _wsystem(const wchar_t *command);`
	Description
	Issues an operating system command.
	system invokes the operating system command processor to execute an operating system command, batch file, or other program named by the string command, from inside an executing C program.
	To be located and executed, the program must be in the current directory or in one of the directories listed in the PATH string in the environment.
	The COMSPEC environment variable is used to find the command processor program, so it need not be in the current directory.
	Return... more (see page 1136)

ultoa, _ultow (see page 1137)	**Header File**
	stdlib.h
	Category
	Conversion Routines, Math Routines
	Prototype
	`char *ultoa(unsigned long value, char *string, int radix);`
	`wchar_t *_ultow(unsigned long value, wchar_t *string, int radix);`
	Description
	Converts an **unsigned long** to a string.
	ultoa converts value to a null-terminated string and stores the result in string. value is an **unsigned long**.
	radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. ultoa performs no overflow checking, and if value is negative and radix equals 10, it does not set the minus sign.
	Note: The space allocated for string must be... more (see page 1137)

wcstombs (see page 1137)	**Header File** stdlib.h **Category** Conversion Routines, Memory and String Manipulation Routines **Prototype** `size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);` **Description** Converts a **wchar_t** array into a multibyte string. wcstombs converts the type **wchar_t** elements contained in pwcs into a multibyte character string s. The process terminates if either a null character or an invalid multibyte character is encountered. No more than n bytes are modified. If n number of bytes are processed before a null character is reached, the array s is not null terminated. The behavior of wcstombs is affected by the setting of LC_CTYPE category of the... more (see page 1137)
wctomb (see page 1138)	**Header File** stdlib.h **Category** Conversion Routines, Memory and String Manipulation Routines **Prototype** `int wctomb(char *s, wchar_t wc);` **Description** Converts **wchar_t** code to a multibyte character. If s is not null, wctomb determines the number of bytes needed to represent the multibyte character corresponding to wc (including any change in shift state). The multibyte character is stored in s. At most MB_CUR_MAX characters are stored. If the value of wc is zero, wctomb is left in the initial state. The behavior of wctomb is affected by the setting of LC_CTYPE category of the current locale. **Return Value** If s is a... more (see page 1138)

3.1.4.29.1 EXIT_xxxx #defines

Header File

stdlib.h

Description

Constants defining exit conditions for calls to the exit function.

Name	Meaning
EXIT_SUCCESS	Normal program termination
EXIT_FAILURE	Abnormal program termination

3.1.4.29.2 RAND_MAX #define

Header File

stdlib.h

Syntax

Description

Maximum value returned by rand function.

3.1.4.29.3 _argc

Header File

stdlib.h

Syntax

```
extern int _argc;
```

Description

_argc has the same value as argc (passed to main) when the program starts. This variable holds the number of arguments passed to the program. The value includes the name of the program itself, so _argc and argc are always at least 1.

3.1.4.29.4 _argv, _wargv

Header File

stdlib.h

Syntax

```
extern char **_argv;
extern wchar_t ** _wargv
```

Description

_argv points to an array containing the original command-line arguments (the elements of argv[]) passed to main when the program starts.

_wargv is the Unicode version of _argv.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_argv		+		
_wargv		NT only		

3.1.4.29.5 _atoi64, _wtoi64

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
__int64 _atoi64(const char *s);
__int64 _wtoi64(const wchar_t *s);
```

Description

Converts a string to an __**int64**.

The syntax of the string must be:

```
__int64 ::= [isspace]* [sign] digit [digit]*
```

Only decimal integers are acceptable.

_wtoi64 is the wide-character version. It converts a wide-character string to an __**int64**.

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in atoi (results are undefined). There is no defined method to return an error indication to the caller. The result is undefined if the input string is invalid.

Return Value

Returns the converted value of the input string. If the string cannot be converted to a __**int64**, the return value is 0.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_atoi64		+		
_wtoi64		+		

3.1.4.29.6 _crotl, _crotr

Header File

stdlib.h

Category

Math Routines

Prototype

```
unsigned char _crotl(unsigned char val, int count);
unsigned char _crotr(unsigned char val, int count);
```

Description

Rotates an unsigned char left or right.

_crotl rotates the given val to the left count bits. _crotr rotates the given val to the right count bits.

The argument val is an **unsigned char**, or its equivalent in decimal or hexadecimal form.

Return Value

- The functions return the rotated byte:
- _crotl returns the value of val left-rotated count bits.
- _crotr returns the value of val right-rotated count bits.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.7 _exit

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
void _exit(int status);
```

Description

Terminates program.

_exit terminates execution without closing any files, flushing any output, or calling any exit functions.

The calling process uses status as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error.

Return Value

None.

Example

```
#include   <stdlib.h>
#include   <stdio.h>
void done(void);
int main(void)
{
   atexit(done);
   _exit(0);
   return 0;
}
void done()
```

```
{
   printf("hello\n");
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.29.8 _fullpath, _wfullpath

Header File

stdlib.h

Category

Directory Control Routines

Prototype

```
char * _fullpath(char *buffer, const char *path, int buflen);
```

```
wchar_t * _wfullpath(wchar_t *buffer, const wchar_t *path, int buflen);
```

Description

Converts a path name from relative to absolute.

_fullpath converts the relative path name in path to an absolute path name that is stored in the array of characters pointed to by buffer. The maximum number of characters that can be stored at buffer is buflen. The function returns NULL if the buffer isn't big enough to store the absolute path name or if the path contains an invalid drive letter.

If buffer is NULL, _fullpath allocates a buffer of up to _MAX_PATH characters. This buffer should be freed using free when it is no longer needed. _MAX_PATH is defined in stdlib.h.

Return Value

If successful the _fullpath function returns a pointer to the buffer containing the absolute path name.

On error, this function returns NULL.

Example

```
#include <stdio.h>
#include <stdlib.h>
char buf[_MAX_PATH];
void main(int argc, char *argv[])
{
  for ( ; argc; argv++, argc--)
  {
    if (_fullpath(buf, argv[0], _MAX_PATH) == NULL)
      printf("Unable to obtain full path of %s\n",argv[0]);
    else
      printf("Full path of %s is %s\n",argv[0],buf);
  }
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_fullpath		+		

_wfullpath		NT only		

3.1.4.29.9 _lrand

Header File

stdlib.h

Category

Math Routines

Prototype

```
long _lrand(void);
```

Description

_lrand is the long random number generator function. _rand uses a multiplicative congruential random number generator with period 2^64 to return successive pseudo-random numbers in the range from 0 to 2^31 - 1.

The generator is reinitialized by calling srand with an argument value of 1. It can be set to a new starting point by calling srand with a given seed number.

3.1.4.29.10 _lrotl, _lrotr

Header File

stdlib.h

Category

Math Routines

Prototype

```
unsigned long _lrotl(unsigned long val, int count);
unsigned long _lrotr(unsigned long val, int count);
```

Description

Rotates an **unsigned long** integer value to the left or right.

_lrotlrotates the given val to the left count bits. _lrotr rotates the given val to the right count bits.

Return Value

The functions return the rotated integer:

- _lrotl returns the value of val left-rotated count bits.
- _lrotr returns the value of val right-rotated count bits.

Example

```
#include <stdlib.h>
#include <stdio.h>
/* function prototypes */
int lrotl_example(void);
int lrotr_example(void);
/* lrotl example */
int lrotl_example(void)
```

```
{
   unsigned long result;
   unsigned long value = 100;
   result = _lrotl(value,1);
   printf("The value %lu rotated left one bit is: %lu\n", value, result);
   return 0;
}
/* lrotr example */
int lrotr_example(void)
{
   unsigned long result;
   unsigned long value = 100;
   result = _lrotr(value,1);
   printf("The value %lu rotated right one bit is: %lu\n", value, result);
   return 0;
}
int main(void)
{
   lrotl_example();
   lrotr_example();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.11 _makepath, _wmakepath

Header File

stdlib.h

Category

Directory Control Routines

Prototype

```
void _makepath(char *path, const char *drive, const char *dir, const char *name, const char *ext);
```

```
void _wmakepath(wchar_t *path, const wchar_t *drive, const wchar_t *dir, const wchar_t *name, const wchar_t *ext);
```

Description

Builds a path from component parts.

_makepath makes a path name from its components. The new path name is

```
X:\DIR\SUBDIR\NAME.EXT
```

where

drive	=	X:
dir	=	\DIR\SUBDIR\
name	=	NAME
ext	=	.EXT

If drive is empty or NULL, no drive is inserted in the path name. If it is missing a trailing colon (:), a colon is inserted in the path name.		
If dir is empty or NULL, no directory is inserted in the path name. If it is missing a trailing slash (\ or /), a backslash is inserted in the path name.		
If name is empty or NULL, no file name is inserted in the path name.		
If ext is empty or NULL, no extension is inserted in the path name. If it is missing a leading period (.), a period is inserted in the path name.		
_makepath assumes there is enough space in path for the constructed path name. The maximum constructed length is _MAX_PATH. _MAX_PATH is defined in stdlib.h.		
_makepath and _splitpath are invertible; if you split a given path with _splitpath, then merge the resultant components with _makepath, you end up with path.		

If drive is empty or NULL, no drive is inserted in the path name. If it is missing a trailing colon (:), a colon is inserted in the path name.

If dir is empty or NULL, no directory is inserted in the path name. If it is missing a trailing slash (\ or /), a backslash is inserted in the path name.

If name is empty or NULL, no file name is inserted in the path name.

If ext is empty or NULL, no extension is inserted in the path name. If it is missing a leading period (.), a period is inserted in the path name.

_makepath assumes there is enough space in path for the constructed path name. The maximum constructed length is _MAX_PATH. _MAX_PATH is defined in stdlib.h.

_makepath and _splitpath are invertible; if you split a given path with _splitpath, then merge the resultant components with _makepath, you end up with path.

Return Value

Example

```
#include <dir.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  char s[_MAX_PATH];
  char drive[_MAX_DRIVE];
  char dir[_MAX_DIR];
  char file[_MAX_FNAME];
  char ext[_MAX_EXT];
  getcwd(s,_MAX_PATH);           /* get current working directory */
 if (s[strlen(s)-1] != '\\')
     strcat(s,"\\");             /* append a trailing \ character */
  _splitpath(s,drive,dir,file,ext); /* split the string to separate
    elems */
  strcpy(file,"DATA");
  strcpy(ext,".TXT");
  _makepath(s,drive,dir,file,ext); /* merge everything into one string */
  puts(s);                       /* display resulting string */
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_makepath		+		
_wmakepath		+		

3.1.4.29.12 _rotl, _rotr

Header File

stdlib.h

Category

Math Routines

Prototype

```
unsigned short _rotl(unsigned short value, int count);
unsigned short _rotr(unsigned short value, int count);
```

Description

Bit-rotates an **unsigned** short integer value to the left or right.

_rotl rotates the given value to the left count bits.

_rotr rotates the given value to the right count bits.

Return Value

_rotl, and _rotr return the rotated integer:

- _rotl returns the value of value left-rotated count bits.
- _rotr returns the value of value right-rotated count bits.

Example

```
#include <stdlib.h>
#include <stdio.h>
/* rotl example */
int rotl_example(void)
{
   unsigned value, result;
   value = 32767;
   result = _rotl(value, 1);
   printf("The value %u rotated left one bit is: %u\n", value, result);
   return 0;
}
/* rotr example */
int rotr_example(void)
{
   unsigned value, result;
   value = 32767;
   result = _rotr(value, 1);
   printf("The value %u rotated right one bit is: %u\n", value, result);
   return 0;
}
int main(void)
{
   rotl_example();
   rotr_example();
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_rotl		+		
_rotr				

3.1.4.29.13 _searchenv, _wsearchenv

Header File

stdlib.h

Category

Miscellaneous Routines

Prototype

```
char *_searchenv(const char *file, const char *varname, char *buf);
```

```
char *_wsearchenv(const wchar_t *file, const wchar_t *varname, wchar_t *buf);
```

Description

Looks for a file, using an environment variable as the search path.

_searchenv attempts to locate file, searching along the path specified by the operating system environment variable varname. Typical environment variables that contain paths are PATH, LIB, and INCLUDE.

_searchenv searches for the file in the current directory of the current drive first. If the file is not found there, the environment variable varname is fetched, and each directory in the path it specifies is searched in turn until the file is found, or the path is exhausted.

When the file is located, the full path name is stored in the buffer pointed to by buf. This string can be used in a call to access the file (for example, with fopen or exec...). The buffer is assumed to be large enough to store any possible file name. If the file cannot be successfully located, an empty string (consisting of only a null character) will be stored at buf.

Return Value

None.

Example

```
#include <stdio.h>
#include <stdlib.h>

char buf[_MAX_PATH];

int main(void)
{
  /* ILINK32 will be found in your installation directory */
  _searchenv("ILINK32.EXE","PATH",buf);
  if (buf[0] == '\0')
    printf("ILINK32.EXE not found\n");
  else
    printf("ILINK32.EXE found in %s\n", buf);

  /* looks for nonexistent file */
  _searchenv("NOTEXIST.FIL","PATH",buf);
  if (buf[0] == '\0')
    printf("NOTEXIST.FIL not found\n");
  else
```

```
    printf("NOTEXIST.FIL found in %s\n", buf);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_searchenv		+		
_wsearchenv		NT only		

3.1.4.29.14 _searchstr, _wsearchstr

Header File

stdlib.h

Category

Miscellaneous Routines

Prototype

```
void _searchstr(const char *file, const char *ipath, char *buf);
```

```
void _wsearchstr(const wchar_t *file, const wchar_t *ipath,wchar_t *pathname);
```

Description

Searches a list of directories for a file.

_searchstr attempts to locate file, searching along the path specified by the string ipath.

_searchstr searches for the file in the current directory of the current drive first. If the file is not found there, each directory in ipath is searched in turn until the file is found, or the path is exhausted. The directories in ipath must be separated by semicolons.

When the file is located, the full path name is stored in the buffer pointed by by buf. This string can be used in a call to access the file (for example, with fopen or exec...). The buffer is assumed to be large enough to store any possible file name. The constant _MAX_PATH defined in stdlib.h, is the size of the largest file name. If the file cannot be successfully located, an empty string (consisting of only a null character) will be stored at buf.

Return Value

None.

Example

```
#include <stdio.h>
#include <stdlib.h>
char buf[_MAX_PATH];
int main(void)
{
  /* look for ILINK32.EXE */
  _searchstr("ILINK32.EXE", "PATH", buf);
  if (buf[0] == '\0')
  printf ("ILINK32.EXE not found\n");
  else
  printf ("ILINK32.EXE found in %s\n", buf);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_searchstr		+		
_wsearchstr		NT only		

3.1.4.29.15 _splitpath, _wsplitpath

Header File

stdlib.h

Category

Directory Control Routines

Prototype

```
void _splitpath(const char *path, char *drive, char *dir, char *name, char *ext);
```

```
void _wsplitpath(const wchar_t *path, wchar_t *drive, wchar_t *dir, wchar_t *name, wchar_t *ext);
```

Description

Splits a full path name into its components.

_splitpath takes a file's full path name (path) as a string in the form

X:\DIR\SUBDIR\NAME.EXT

and splits path into its four components. It then stores those components in the strings pointed to by drive, dir, name, and ext. (All five components must be passed, but any of them can be a null, which means the corresponding component will be parsed but not stored.) The maximum sizes for these strings are given by the constants _MAX_DRIVE, _MAX_DIR, _MAX_PATH, _MAX_FNAME, and _MAX_EXT (defined in stdlib.h), and each size includes space for the null-terminator. These constants are defined in stdlib.h.

_MAX_PATH	path
_MAX_DRIVE	drive; includes colon (:)
_MAX_DIR	dir; includes leading and trailing backslashes (\)
_MAX_FNAME	name
_MAX_EXT	ext; includes leading dot (.)

_splitpath assumes that there is enough space to store each non-null component.

When _splitpath splits path, it treats the punctuation as follows:

- drive includes the colon (C:, A:, and so on).
- dir includes the leading and trailing backslashes (\BC\include\, \source\, and so on).
- name includes the file name.
- ext includes the dot preceding the extension (.C, .EXE, and so on).

_makepath and _splitpath are invertible; if you split a given path with _splitpath, then merge the resultant components with _makepath, you end up with path.

Return Value

None.

Example

```
#include <dir.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  char s[_MAX_PATH];
  char drive[_MAX_DRIVE];
  char dir[_MAX_DIR];
  char file[_MAX_FNAME];
  char ext[_MAX_EXT];
  /* get current working directory */
  getcwd(s,_MAX_PATH);
  if (s[strlen(s)-1] != '\\')
  /* append a trailing \ character */
    strcat(s,"\\");
  /* split the string to separate elems */
  _splitpath(s,drive,dir,file,ext);
  strcpy(file,"DATA");
  strcpy(ext,".TXT");
  /* merge everything into one string */
  _makepath(s,drive,dir,file,ext);
  /* display resulting string */
  puts(s);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_splitpath		+		
_wsplitpath		+		

3.1.4.29.16 _ui64toa, _ui64tow

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *_ui64toa(unsigned __int64 value, char *string, int radix);
```

```
wchar_t *_ui64tow(unsigned __int64 value, wchar_t *string, int radix);
```

Description

_ui64toa converts an unsigned __**int64** to a string.

_ui64tow is the unicode version. _ui64tow converts an unsigned __**int64** to a wide-character string.

These functions convert value to a null-terminated string and store the result in string. value is an __**int64**.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus sign (**-**).

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character

(\0). Can return up to 33 bytes.

Return Value

Returns a pointer to string.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_ui64toa				
_ui64tow		+		

3.1.4.29.17 abort

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
void abort(void);
```

Description

Abnormally terminates a program.

abort causes an abnormal program termination by calling raise(SIGABRT). If there is no signal handler for SIGABRT, then abort writes a termination message (Abnormal program termination) on stderr, then aborts the program by a call to _exit with exit code 3.

Return Value

abort returns the exit code 3 to the parent process or to the operating system command processor.

Example

```
#include <stdio.h>
#include <stdlib.h>
 int main(void)
 {
   printf("Calling abort()\n");
   abort();
   return 0; /* This is never reached */
 }
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.18 atexit

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
int atexit(void (_USERENTRY * func)(void));
```

Description

Registers termination function.

atexit registers the function pointed to by func as an exit function. Upon normal termination of the program, exit calls func just before returning to the operating system. fcmp must be used with the _USERENTRY calling convention.

Each call to atexit registers another exit function. Up to 32 functions can be registered. They are executed on a last-in, first-out basis (that is, the last function registered is the first to be executed).

Return Value

atexit returns 0 on success and nonzero on failure (no space left to register the function).

Example

```
#include <stdio.h>
#include <stdlib.h>
void exit_fn1(void)
{
   printf("Exit function #1 called\n");
}
void exit_fn2(void)
{
   printf("Exit function #2 called\n");
}
int main(void)
{
   /* post exit function #1 */
   atexit(exit_fn1);
   /* post exit function #2 */
   atexit(exit_fn2);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.19 atoi, _wtoi

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
int atoi(const char *s);
```

```
int _wtoi(const wchar_t *s);
```

Description

Converts a string to an integer.

- atoi converts a string pointed to by s to **int**; atoi recognizes (in the following order)
- An optional string of tabs and spaces
- An optional sign
- A string of digits

The characters must match this generic format:

[ws] [sn] [ddd]

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in atoi (results are undefined).

Return Value

atoi returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (**int**), atoi returns 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   int n;
   char *str = "12345.67";
   n = atoi(str);
   printf("string = %s integer = %d\n", str, n);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
atoi	+	+	+	+
_wtoi		+		

3.1.4.29.20 atol, _wtol

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
long atol(const char *s);
```

```
long _wtol(const wchar_t *s);
```

Description

Converts a string to a long.

- atol converts the string pointed to by s to **long**. atol recognizes (in the following order)
- An optional string of tabs and spaces

- An optional sign
- A string of digits

The characters must match this generic format:

```
[ws] [sn] [ddd]
```

In this function, the first unrecognized character ends the conversion. There are no provisions for overflow in atol (results are undefined).

Return Value

atol returns the converted value of the input string. If the string cannot be converted to a number of the corresponding type (b), atol returns 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   long l;
   char *lstr = "98765432";
   l = atol(lstr);
   printf("string = %s integer = %ld\n", lstr, l);
   return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
atol	+	+	+	+
_wtol		+		

3.1.4.29.21 bsearch

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void *bsearch(const void *key, const void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Binary search of an array.

bsearch searches a table (array) of nelem elements in memory, and returns the address of the first entry in the table that matches the search key. The array must be in order. If no match is found, bsearch returns 0.

Note: Because this is a binary search, the first matching entry is not necessarily the first entry in the table.

The type size_t is defined in stddef.h header file.

- nelem gives the number of elements in the table.
- width specifies the number of bytes in each table entry.

The comparison routine fcmp must be used with the _USERENTRY calling convention.

fcmp is called with two arguments: elem1 and elem2. Each argument points to an item to be compared. The comparison function compares each of the pointed-to items (*elem1 and *elem2), and returns an integer based on the results of the comparison.

- For bsearch, the fcmp return value is
- < 0 if *elem1 < *elem2
- == 0 if *elem1 == *elem2
- > 0 if *elem1 > *elem2

Return Value

bsearch returns the address of the first entry in the table that matches the search key. If no match is found, bsearch returns 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
typedef int (*fptr)(const void*, const void*);
#define NELEMS(arr) (sizeof(arr) / sizeof(arr[0]))
int numarray[] = {123, 145, 512, 627, 800, 933};
int numeric (const int *p1, const int *p2)
{
   return(*p1 - *p2);
}
#pragma argsused
int lookup(int key)
{
   int  *itemptr;
   /* The cast of (int(*)(const void *,const void*))
      is needed to avoid a type mismatch error at
      compile time */
   itemptr = (int *) bsearch (&key, numarray, NELEMS(numarray),
      sizeof(int), (fptr)numeric);
   return (itemptr != NULL);
}
int main(void)
{
   if (lookup(512))
      printf("512 is in the table.\n");
   else
      printf("512 isn't in the table.\n");
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.22 div

Header File

stdlib.h

Category

Math Routines

Prototype

```
div_t div(int numer, int denom);
```

Description

Divides two integers, returning quotient and remainder.

div divides two integers and returns both the quotient and the remainder as a div_t type. numer and denom are the numerator and denominator, respectively. The div_t type is a structure of integers defined (with **typedef**) in stdlib.h as follows:

```
typedef struct {
int quot; /* quotient */
int rem; /* remainder */
} div_t;
```

Return Value

div returns a structure whose elements are quot (the quotient) and rem (the remainder).

Example

```
/* div example */
#include <stdlib.h>
#include <stdio.h>
div_t x;
int main(void)
{
   x = div(10,3);
   printf("10 div 3 = %d remainder %d\n",
   x.quot, x.rem);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.23 ecvt

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *ecvt(double value, int ndig, int *dec, int *sign);
```

Description

Converts a floating-point number to a string.

ecvt converts value to a null-terminated string of ndig digits, starting with the leftmost significant digit, and returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through dec (a negative value for dec means that the decimal lies to the left of the returned digits). There is no decimal point in the string itself. If the sign of value is negative, the word pointed to by sign is nonzero; otherwise, it's 0. The low-order digit is rounded.

Return Value

The return value of ecvt points to static data for the string of digits whose content is overwritten by each call to ecvt and fcvt.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   char *string;
   double value;
   int dec, sign;
   int ndig = 10;
   value = 9.876;
   string = ecvt(value, ndig, &dec, &sign);
   printf("string = %s      dec = %d sign = %d\n", string, dec, sign);
   value = -123.45;
   ndig= 15;
   string = ecvt(value,ndig,&dec,&sign);
   printf("string = %s dec = %d sign = %d\n", string, dec, sign);
   value = 0.6789e5; /* scientific notation */
   ndig = 5;
   string = ecvt(value,ndig,&dec,&sign);
   printf("string = %s         dec = %d sign = %d\n", string, dec, sign);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.24 _environ, _wenviron

Header File

stdlib.h

Syntax

```
extern char ** _environ;
```

```
extern wchar_t ** _wenviron
```

Description

_environ is an array of pointers to strings; it is used to access and alter the operating system environment variables. Each string is of the form:

```
envvar = varvalue
```

where envvar is the name of an environment variable (such as PATH), and varvalue is the string value to which envvar is set (such as C:\Utils;C:\MyPrograms). The string varvalue can be empty.

When a program begins execution, the operating system environment settings are passed directly to the program. Note that env, the third argument to main, is equal to the initial setting of _environ.

The _environ array can be accessed by getenv; however, the putenv function is the only routine that should be used to add, change or delete the _environ array entries. This is because modification can resize and relocate the process environment array, but _environ is automatically adjusted so that it always points to the array.

Portability

	POSIX	Win32	ANSI C	ANSI C++
_environ		+		
_wenviron		NT only		

3.1.4.29.25 exit

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
void exit(int status);
```

Description

Terminates program.

exit terminates the calling process. Before termination, all files are closed, buffered output (waiting to be output) is written, and any registered "exit functions" (posted with atexit) are called.

status is provided for the calling process as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error. It can be, but is not required, to be set with one of the following:

EXIT_FAILURE	Abnormal program termination; signal to operating system that program has terminated with an error
EXIT_SUCCESS	Normal program termination

Return Value

None.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   int status;
   printf("Enter either 1 or 2\n");
   status = getchar();
   exit(status - '0');
/* Note: this line is never reached */
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.26 fcvt

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *fcvt(double value, int ndig, int *dec, int *sign);
```

Description

Converts a floating-point number to a string.

fcvt converts value to a null-terminated string digit starting with the leftmost significant digit with ndig digits to the right of the decimal point. fcvt then returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through dec (a negative value for dec means to the left of the returned digits). There is no decimal point in the string itself. If the sign of value is negative the word pointed to by sign is nonzero; otherwise it is 0.

The correct digit has been rounded for the number of digits to the right of the decimal point specified by ndig.

Return Value

The return value of fcvt points to static data whose content is overwritten by each call to fcvt and ecvt.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
  char *str;
  double num;
  int dec, sign, ndig = 5;
  /* a regular number */
  num = 9.876;
  str = fcvt(num, ndig, &dec, &sign);
  printf("string = %10s decimal place = %d sign = %d\n", str, dec, sign);
  /* a negative number */
  num = -123.45;
  str = fcvt(num, ndig, &dec, &sign);
  printf("string = %10s decimal place = %d sign = %d\n", str, dec, sign);
  /* scientific notation */
  num = 0.678e5;
  str = fcvt(num, ndig, &dec, &sign);
  printf("string = %10s  decimal place= %d  sign = %d\n", str, dec, sign);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.27 gcvt

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *gcvt(double value, int ndec, char *buf);
```

Description

Converts floating-point number to a string.

gcvt converts value to a null-terminated ASCII string and stores the string in buf. It produces ndec significant digits in FORTRAN F format, if possible; otherwise, it returns the value in the printf E format (ready for printing). It might suppress trailing zeros.

Return Value

gcvt returns the address of the string pointed to by buf.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   char str[25];
   double num;
   int sig = 5; /* significant digits */
   /* a regular number */
   num = 9.876;
   gcvt(num, sig, str);
   printf("string = %s\n", str);
   /* a negative number */
   num = -123.4567;
   gcvt(num, sig, str);
   printf("string = %s\n", str);
   /* scientific notation */
   num = 0.678e5;
   gcvt(num, sig, str);
   printf("string = %s\n", str);
   return(0);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.28 getenv, _wgetenv

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
char *getenv(const char *name);
```

```
wchar_t *_wgetenv(const wchar_t *name);
```

Description

Find or delete an environment variable from the system environment.

The environment consists of a series of entries that are of the form name=string\0.

getenv returns the value of a specified variable. name can be either uppercase or lowercase. name must not include the equal sign (**=**). If the specified environment variable does not exist, getenv returns a NULL pointer.

To delete the variable from the environment, use getenv("name=").

Note: Environment entries must not be changed directly. If you want to change an environment value, you must use putenv.

Return Value

On success, getenv returns the value associated with name.

If the specified name is not defined in the environment, getenv returns a NULL pointer.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
   char *path, *ptr;
   int i = 0;
   /* get the current path environment */
   ptr = getenv("PATH");
   /* set up new path */
   path = (char *) malloc(strlen(ptr)+15);
   strcpy(path,"PATH=");
   strcat(path,ptr);
   strcat(path,";c:\\temp");
   /* replace the current path and display current environment */
   putenv(path);
   while (_environ[i])
       printf("%s\n",_environ[i++]);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
getenv	+	+	+	+
_wgetenv		+		

3.1.4.29.29 itoa, _itow

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *itoa(int value, char *string, int radix);
```

```
wchar_t *_itow(int value, wchar_t *string, int radix);
```

Description

Converts an integer to a string.

itoa converts value to a null-terminated string and stores the result in string. With itoa, value is an integer. _itow is the unicode version of the function. It converts an integer to a wide-character string.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus sign (-).

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character (\0). itoa can return up to 33 bytes.

Return Value

itoa returns a pointer to string.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   int number = 12345;
   char string[25];
   itoa(number, string, 10);
   printf("integer = %d string = %s\n", number, string);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.30 labs

Header File

stdlib.h

Category

Math Routines

Prototype

```
long labs(long int x);
```

Description

Gives long absolute value.

labs computes the absolute value of the parameter x.

Return Value

labs returns the absolute value of x.

Example

```
#include <stdio.h>
#include <math.h>
int main(void)
{
   long result;
   long x = -12345678L;
   result= labs(x);
   printf("number: %ld abs value: %ld\n", x, result);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.31 lfind

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void *lfind(const void *key, const void *base, size_t *num, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Performs a linear search.

lfind makes a linear search for the value of key in an array of sequential records. It uses a user-defined comparison routine fcmp. The fcmp function must be used with the _USERENTRY calling convention.

The array is described as having *num records that are width bytes wide, and begins at the memory location pointed to by base.

Return Value

lfind returns the address of the first entry in the table that matches the search key. If no match is found, lfind returns NULL. The comparison routine must return 0 if *elem1 == *elem2, and nonzero otherwise (elem1 and elem2 are its two parameters).

Example

```
#include <stdio.h>
#include <stdlib.h>
int compare(int *x, int *y)
{
   return( *x - *y );
}
int main(void)
{
   int array[5] = {35, 87, 46, 99, 12};
   size_t nelem = 5;
   int key;
   int *result;
   key = 99;
   result = (int *) lfind(&key, array, &nelem,
      sizeof(int), (int(*)(const void *,const void *))compare);
   if (result)
      printf("Number %d found\n",key);
   else
      printf("Number %d not found\n",key);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.32 lsearch

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void *lsearch(const void *key, void *base, size_t *num, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Performs a linear search.

lsearch searches a table for information. Because this is a linear search, the table entries do not need to be sorted before a call to lsearch. If the item that key points to is not in the table, lsearch appends that item to the table.

- base points to the base (0th element) of the search table.
- num points to an integer containing the number of entries in the table.
- width contains the number of bytes in each entry.
- key points to the item to be searched for (the search key).

The function fcmp must be used with the _USERENTRY calling convention.

The argument fcmp points to a user-written comparison routine, that compares two items and returns a value based on the comparison.

To search the table, lsearch makes repeated calls to the routine whose address is passed in fcmp.

On each call to the comparison routine, lsearch passes two arguments:

key	a pointer to the item being searched for
elem	pointer to the element of base being compared.

fcmp is free to interpret the search key and the table entries in any way.

Return Value

lsearch returns the address of the first entry in the table that matches the search key.

If the search key is not identical to *elem, fcmp returns a nonzero integer. If the search key is identical to *elem, fcmp returns 0.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>     /* for strcmp declaration */
/* initialize number of colors */
char *colors[10] = { "Red", "Blue", "Green" };
int ncolors = 3;
int colorscmp(char **arg1, char **arg2)
{
   return(strcmp(*arg1, *arg2));
}
int addelem(char **key)
{
   int oldn = ncolors;
   lsearch(key, colors, (size_t *)&ncolors, sizeof(char *),
```

```
      (int(*)(const void *,const void *))colorscmp);
   return(ncolors == oldn);
}
int main(void)
{
   int i;
   char *key = "Purple";
   if (addelem(&key))
      printf("%s already in colors table\n", key);
   else
   {
      printf("%s added to colors table\n", key);
   }
   printf("The colors:\n");
   for (i = 0; i < ncolors; i++)
      printf("%s\n", colors[i]);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.33 ltoa, _ltoa, _ltow

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char * ltoa(long value, char * string, int radix);
```

```
char *_ltoa(long value, char *string, int radix);
```

```
wchar_t *_ltow(long value, wchar_t *string, int radix);
```

Description

Converts a **long** to a string. _ltow is the unicode version. It converts a **long** to a wide-charater string.

Converts value to a null-terminated string and stores the result in string. value is a **long**.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. If value is negative and radix is 10, the first character of string is the minus sign (**-**).

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character (\0). Can return up to 33 bytes.

Return Value

Returns a pointer to string.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
```

```
    char string[25];
    long value = 123456789L;
    ltoa(value,string,10);
    printf("number = %ld  string = %s\n", value, string);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ltoa		+		
_ltoa		+		
_ltow		+		

3.1.4.29.34 max

Header File

stdlib.h

Category

C++ Prototyped Routines

Prototype

```
(type) max(a, b);
```

```
template <class T> T max( T t1, T t2 ); // C++ only
```

Description

Returns the larger of two values.

The C macro and the C++ template function compare two values and return the larger of the two. Both arguments and the routine declaration must be of the same type.

Return Value

max returns the larger of two values.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.35 mblen

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
int mblen(const char *s, size_t n);
```

Description

Determines the length of a multibyte character.

If s is not null, mblen determines the number of bytes in the multibyte character pointed to by s. The maximum number of bytes examined is specified by n.

The behavior of mblen is affected by the setting of LC_CTYPE category of the current locale.

Return Value

If s is null, mblen returns a nonzero value if multibyte characters have state-dependent encodings. Otherwise, mblen returns 0.

If s is not null, mblen returns 0 if s points to the null character, and -1 if the next n bytes do not comprise a valid multibyte character; the number of bytes that comprise a valid multibyte character.

Example

```
#include <stdlib.h>
#include <stdio.h>
void main (void)
{
  int i ;
  char *mulbc = (char *)malloc( sizeof( char) );
  wchar_t widec = L'a';
  printf (" convert a wide character to multibyte character:\n" );
  i = wctomb (mulbc, widec);
  printf( "\tCharacters converted: %u\n", i);
  printf( "\tMultibyte character: %x\n\n", mulbc);
  printf( " Find length--in byte-- of multibyte character:\n");
  i = mblen( mulbc, MB_CUR_MAX);
  printf("\tLength--in bytes--if multiple character: %u\n",i);
  printf("\tWide character: %x\n\n", mulbc);
  printf( " Attempt to find length of a Wide character Null:\n");
  widec = L'\0';
  wctomb(mulbc, widec);
  i = mblen( mulbc, MB_CUR_MAX);
  printf("\tLength--in bytes--if multiple character: %u\n",i);
  printf("\tWide character: %x\n\n", mulbc);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.36 mbstowcs

Header File

stdlib.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Description

Converts a multibyte string to a **wchar_t** array.

The function converts the multibyte string s into the array pointed to by pwcs. No more than n values are stored in the array. If an invalid multibyte sequence is encountered, mbstowcs returns (size_t) -1.

The pwcs array will not be terminated with a zero value if mbstowcs returns n.

Return Value

If an invalid multibyte sequence is encountered, mbstowcs returns (size_t) -1. Otherwise, the function returns the number of array elements modified, not including the terminating code, if any.

Example

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  int x;
  char    *mbst = (char *)malloc(MB_CUR_MAX);
  wchar_t *pwst = L"Hi";
  wchar_t *pwc      = (wchar_t *)malloc(sizeof( wchar_t));
  printf ("Convert to multibyte string:\n");
  x = wcstombs (mbst, pwst, MB_CUR_MAX);
  printf ("\tCharacters converted %u\n",x);
  printf ("\tHEx value of first");
  printf (" multibyte character: %#.4x\n\n", mbst);
  printf ("Convert back to wide character string:\n");
  x = mbstowcs(pwc, mbst, MB_CUR_MAX);
  printf( "\tCharacters converted: %u\n",x);
  printf( "\tHex value of first");
  printf( "wide character: %#.4x\n\n", pwc);
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.37 mbtowc

Header File

stdlib.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

Converts a multibyte character to **wchar_t** code.

If s is not null, mbtowc determines the number of bytes that comprise the multibyte character pointed to by s. Next, mbtowc determines the value of the type **wchar_t** that corresponds to that multibyte character. If there is a successful match between **wchar_t** and the multibyte character, and pwc is not null, the **wchar_t** value is stored in the array pointed to by pwc. At most n characters are examined.

Return Value

When s points to an invalid multibyte character, -1 is returned. When s points to the null character, 0 is returned. Otherwise,

mbtowc returns the number of bytes that comprise the converted multibyte character.

The return value never exceeds MB_CUR_MAX or the value of n.

Example

```
#include <stdlib.h>
#include<stdio.h>
void main(void)
{
  int     x;
  char     *mbchar    = (char *)calloc(1, sizeof( char));
  wchar_t  wchar      = L'a';
  wchar_t   *pwcnull = NULL;
  wchar_t  *pwchar    = (wchar_t *)calloc(1,  sizeof( wchar_t ));
  printf ("Convert a wide character to multibyte character:\n");
  x = wctomb( mbchar, wchar);
  printf( "\tCharacters converted: %u\n", x);
  printf( "\tMultibyte character: %x\n\n", mbchar);
  printf ("Convert multibyte character back to a wide character:\n");
  x = mbtowc( pwchar, mbchar, MB_CUR_MAX );
  printf( "\tBytes converted: %u\n", x);
  printf( "\tWide character: %x\n\n", pwchar);
  printf ("Atempt to convert when target is NULL\n" );
  printf (" returns the length of the multibyte character:\n" );
  x = mbtowc (pwcnull, mbchar, MB_CUR_MAX );
  printf ( "\tlength of multibyte character:%u\n\n", x );
  printf ("Attempt to convert a NULL pointer to a" );
  printf (" wide character:\n" );
  mbchar = NULL;
  x = mbtowc (pwchar, mbchar, MB_CUR_MAX);
  printf( "\tBytes converted: %u\n", x );
  }
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.38 min

Header File

stdlib.h

Category

C++ Prototyped Routines

Prototype

```
(type) min(a, b); /* macro version */
```

```
template <class T> T min( T t1, T t2 );// C++ only
```

Description

Returns the smaller of two values.

The C macro and the C++ template function compare two values and return the smaller of the two. Both arguments and the routine declaration must be of the same type.

Return Value

min returns the smaller of two values.

3.1.4.29.39 putenv, _wputenv

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
int putenv(const char *name);
```

```
int _wputenv(const wchar_t *name);
```

Description

Adds string to current environment.

putenv accepts the string name and adds it to the environment of the current process. For example,

```
putenv(“PATH=C:\\BC”);
```

putenv can also be used to modify an existing name. name can be either uppercase or lowercase. name must not include the equal sign (**=**). You can set a variable to an empty value by specifying an empty string on the right side of the ‘=’ sign.

putenv can be used only to modify the current program’s _environment. Once the program ends, the old _environment is restored. The _environment of the current process is passed to child processes, including any changes made by putenv.

Note that the string given to putenv must be static or global. Unpredictable results will occur if a local or dynamic string given to putenv is used after the string memory is released.

Return Value

On success, putenv returns 0; on failure, -1.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <string.h>
int main(void)
{
   char *path, *ptr;
   int i = 0;
   /* get the current path environment */
   ptr = getenv("PATH");
   /* set up new path */
   path = (char *) malloc(strlen(ptr)+15);
   strcpy(path,"PATH=");
   strcat(path,ptr);
   strcat(path,";c:\\temp");
   /* replace the current path and display current environment */
   putenv(path);
   while (_environ[i])
       printf("%s\n",_environ[i++]);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
putenv		+		
_wputenv		NT only		

3.1.4.29.40 qsort

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void qsort(void *base, size_t nelem, size_t width, int (_USERENTRY *fcmp)(const void *, const void *));
```

Description

Sorts using the quicksort algorithm.

qsort is an implementation of the “median of three” variant of the quicksort algorithm. qsort sorts the entries in a table by repeatedly calling the user-defined comparison function pointed to by fcmp.

- base points to the base (0th element) of the table to be sorted.
- nelem is the number of entries in the table.
- width is the size of each entry in the table, in bytes.

fcmp, the comparison function, must be used with the _USERENTRY calling convention.

- fcmp accepts two arguments, elem1 and elem2, each a pointer to an entry in the table. The comparison function compares each of the pointed-to items (*elem1 and *elem2), and returns an integer based on the result of the comparison.
- *elem1 < *elem2 fcmp returns an integer < 0
- *elem1 == *elem2 fcmp returns 0
- *elem1 > *elem2 fcmp returns an integer > 0

In the comparison, the less-than symbol (<) means the left element should appear before the right element in the final, sorted sequence. Similarly, the greater-than (>) symbol means the left element should appear after the right element in the final, sorted sequence.

Return Value

None.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int sort_function( const void *a, const void *b);
char list[5][4] = { "cat", "car", "cab", "cap", "can" };
int main(void)
{
   int  x;
   qsort((void *)list, 5, sizeof(list[0]), sort_function);
   for (x = 0; x < 5; x++)
      printf("%s\n", list[x]);
   return 0;
```

```
}
int sort_function( const void *a, const void *b)
{
   return( strcmp((char *)a,(char *)b) );
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.41 rand

Header File

stdlib.h

Category

Math Routines

Prototype

```
int rand(void);
```

Description

Random number generator.

rand uses a multiplicative congruential random number generator with period 2 to the 32nd power to return successive pseudo-random numbers in the range from 0 to RAND_MAX. The symbolic constant RAND_MAX is defined in stdlib.h.

Return Value

rand returns the generated pseudo-random number.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    int i;
    randomize();
    printf("Ten random numbers from 0 to 99\n\n");
    for(i=0; i<10; i++)
       printf("%d\n", rand() % 100);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.42 random

Header File

stdlib.h

Category

Math Routines

Prototype

```
int random(int num);
```

Description

Random number generator.

random returns a random number between 0 and (num-1). random(num) is a macro defined in stdlib.h. Both num and the random number returned are integers.

Return Value

random returns a number between 0 and (num-1).

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
/* prints a random number in the range 0 to 99 */
int main(void)
 {
    randomize();
    printf("Random number in the 0-99 range: %d\n", random (100));
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.43 randomize

Header File

stdlib.h, time.h

Category

Math Routines

Prototype

```
void randomize(void);
```

Description

Initializes random number generator.

randomize initializes the random number generator with a random value.

Return Value

None.

Example

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <time.h>
int main(void)
{
    int i;
    randomize();
    printf("Ten random numbers from 0 to 99\n\n");
    for(i=0; i<10; i++)
        printf("%d\n", rand() % 100);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.29.44 srand

Header File

stdlib.h

Category

Math Routines

Prototype

```
void srand(unsigned seed);
```

Description

Initializes random number generator.

The random number generator is reinitialized by calling srand with an argument value of 1. It can be set to a new starting point by calling srand with a given seed number.

Return Value

None.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
int main(void)
{
   int i;
   time_t t;
   srand((unsigned) time(&t));
   printf("Ten random numbers from 0 to 99\n\n");
   for(i=0; i<10; i++)
       printf("%d\n", rand() % 100);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.45 strlen, _mbslen, wcslen, _mbstrlen

Header File

string.h, mbstring.h, stdlib.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
size_t strlen(const char *s);
size_t wcslen(const wchar_t *s);
size_t _mbslen(const unsigned char *s);
size_t _mbstrlen(const char *s)
```

Description

Calculates the length of a string.

strlen calculates the length of s.

_mbslen and _mbstrlen test the string argument to determine the number of multibyte characters they contain.

_mbstrlen is affected by the LC_CTYPE category setting as determined by the setlocale function. The function tests to determine whether the string argument is a valid multibyte string.

_mbslen is affected by the code page that is in use. This function doesn't test for multibyte validity.

Return Value

strlen returns the number of characters in s, not counting the null-terminating character.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string = "CodeGear";
   printf("%d\n", strlen(string));
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strlen	+	+	+	+
_mbslen		+		
wcslen		+	+	+
_mbstrlen		+		

3.1.4.29.46 strtod, _strtold, wcstod, _wcstold

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
double strtod(const char *s, char **endptr);
double wcstod(const wchar_t *s, wchar_t **endptr);
long double _strtold(const char *s, char **endptr);
long double _wcstold(const wchar_t *s, wchar_t **endptr);
```

Description

Convert a string to a **double** or **long double** value.

strtod converts a character string, s, to a **double** value. s is a sequence of characters that can be interpreted as a **double** value; the characters must match this generic format:

```
[ws] [sn] [ddd] [.] [ddd] [fmt[sn]ddd]
```

where:

[ws]	=	optional whitespace
[sn]	=	optional sign (+ or -)
[ddd]	=	optional digits
[fmt]	=	optional e or E
[.]	=	optional decimal point

strtod also recognizes +INF and -INF for plus and minus infinity, and +NAN and -NAN for not-a-number.

For example, here are some character strings that strtod can convert to **double**:

```
+ 1231.1981 e-1
502.85E2
+ 2010.952
```

strtod stops reading the string at the first character that cannot be interpreted as an appropriate part of a **double** value.

If endptr is not null, strtod sets *endptr to point to the character that stopped the scan (*endptr = &stopper). endptr is useful for error detection.

_strtold is the l**ong double** version; it converts a string to a **long double** value.

Return Value

These functions return the value of s as a **double** (strtod) or a **long double** (_strtold). In case of overflow, they return plus or minus HUGE_VAL (strtod) or _LHUGE_VAL (_strtold).

Example

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   char input[80], *endptr;
   double value;
   printf("Enter a floating point number:");
   gets(input);
   value = strtod(input, &endptr);
```

```
   printf("The string is %s the number is %lf\n", input, value);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strtod	+	+	+	+
_strtold		+		
wcstod		+	+	+
_wcstold		+		

3.1.4.29.47 strtol, wcstol

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
long strtol(const char *s, char **endptr, int radix);
```

```
long wcstol(const wchar_t *s, wchar_t **endptr, int radix);
```

Description

Converts a string to a **long** value.

strtol converts a character string, s, to a **long** integer value. s is a sequence of characters that can be interpreted as a **long** value; the characters must match this generic format:

```
[ws] [sn] [0] [x] [ddd]
```

where:

[ws]	=	optional whitespace
[sn]	=	optional sign (+ or -)
[0]	=	optional zero (0)
[x]	=	optional x or X
[ddd]	=	optional digits

strtol stops reading the string at the first character it doesn't recognize.

If radix is between 2 and 36, the long integer is expressed in base radix. If radix is 0, the first few characters of s determine the base of the value being converted.

If radix is 1, it is considered to be an invalid value. If radix is less than 0 or greater than 36, it is considered to be an invalid value.

Any invalid value for radix causes the result to be 0 and sets the next character pointer *endptr to the starting string pointer.

If the value in s is meant to be interpreted as octal, any character other than 0 to 7 will be unrecognized.

If the value in s is meant to be interpreted as decimal, any character other than 0 to 9 will be unrecognized.

If the value in s is meant to be interpreted as a number in any other base, then only the numerals and letters used to represent numbers in that base will be recognized. (For example, if radix equals 5, only 0 to 4 will be recognized; if radix equals 20, only 0 to 9 and A to J will be recognized.)

If endptr is not null, strtol sets *endptr to point to the character that stopped the scan (*endptr = &stopper).

Return Value

strtol returns the value of the converted string, or 0 on error.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   char *string = "87654321", *endptr;
   long lnumber;
   /* strtol converts string to long integer  */
   lnumber = strtol(string, &endptr, 10);
   printf("string = %s  long = %ld\n", string, lnumber);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strtol	+	+	+	+
wcstol		+	+	+

3.1.4.29.48 strtoul, wcstoul

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
unsigned long strtoul(const char *s, char **endptr, int radix);
```

```
unsigned long wcstoul(const wchar_t *s, wchar_t **endptr, int radix);
```

Description

Converts a string to an **unsigned long** in the given radix.

strtoul operates the same as strtol, except that it converts a string str to an **unsigned long** value (where strtol converts to a **long**). Refer to the entry for strtol for more information.

Return Value

strtoul returns the converted value, an **unsigned long**, or 0 on error.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
```

```
    char *string = "87654321", *endptr;
    unsigned long lnumber;
    lnumber = strtoul(string, &endptr, 10);
    printf("string = %s  long = %lu\n",
    string, lnumber);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strtoul	+	+	+	+
wcstoul		+	+	+

3.1.4.29.49 swab

Header File

stdlib.h

Category

Memory and String Manipulation Routines

Prototype

```
void swab(char *from, char *to, int nbytes);
```

Description

Swaps bytes.

swab copies nbytes bytes from the from string to the to string. Adjacent even- and odd-byte positions are swapped. This is useful for moving data from one machine to another machine with a different byte order. nbytes should be even.

Return Value

None.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char source[15] = "rFna koBlrna d";
char target[15];
int main(void)
{
    swab(source, target, strlen(source));
    printf("This is target: %s\n", target);
    return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+		

3.1.4.29.50 system, _wsystem

Header File

stdlib.h

Category

Process Control Routines

Prototype

```
int system(const char *command);
```

```
int _wsystem(const wchar_t *command);
```

Description

Issues an operating system command.

system invokes the operating system command processor to execute an operating system command, batch file, or other program named by the string command, from inside an executing C program.

To be located and executed, the program must be in the current directory or in one of the directories listed in the PATH string in the environment.

The COMSPEC environment variable is used to find the command processor program, so it need not be in the current directory.

Return Value

If command is a NULL pointer, system returns nonzero if a command processor is available.

If command is not a NULL pointer, system returns 0 if the command processor was successfully started.

If an error occurred, a -1 is returned and errno is set to one of the following:

ENOENT	Path or file function not found
ENOEXEC	Exec format error
ENOMEM	Not enough memory

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   printf("About to spawn a command-line program.\n");
   system("dir");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
system	+	+		
_wsystem		NT only		

3.1.4.29.51 ultoa, _ultow

Header File

stdlib.h

Category

Conversion Routines, Math Routines

Prototype

```
char *ultoa(unsigned long value, char *string, int radix);
```

```
wchar_t *_ultow(unsigned long value, wchar_t *string, int radix);
```

Description

Converts an **unsigned long** to a string.

ultoa converts value to a null-terminated string and stores the result in string. value is an **unsigned long**.

radix specifies the base to be used in converting value; it must be between 2 and 36, inclusive. ultoa performs no overflow checking, and if value is negative and radix equals 10, it does not set the minus sign.

Note: The space allocated for string must be large enough to hold the returned string, including the terminating null character (\0). ultoa can return up to 33 bytes.

Return Value

ultoa returns string.

Example

```
#include <stdlib.h>
#include <stdio.h>
int main( void )
{
   unsigned long lnumber = 3123456789L;
   char string[25];
   ultoa(lnumber,string,10);
   printf("string = %s  unsigned long = %lu\n",string,lnumber);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ultoa		+		
_ultow		+		

3.1.4.29.52 wcstombs

Header File

stdlib.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Description

Converts a **wchar_t** array into a multibyte string.

wcstombs converts the type **wchar_t** elements contained in pwcs into a multibyte character string s. The process terminates if either a null character or an invalid multibyte character is encountered.

No more than n bytes are modified. If n number of bytes are processed before a null character is reached, the array s is not null terminated.

The behavior of wcstombs is affected by the setting of LC_CTYPE category of the current locale.

Return Value

If an invalid multibyte character is encountered, wcstombs returns (size_t) -1. Otherwise, the function returns the number of bytes modified, not including the terminating code, if any.

Example

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  int x;
  char *pbuf = (char*)malloc( MB_CUR_MAX);
  wchar_t *pwcsEOL = L'\0';
  char *pwchi= L"Hi there!";

  printf (" Convert entire wchar string into a multibyte string:\n");
  x = wcstombs( pbuf, pwchi,MB_CUR_MAX);
  printf ("Character converted: %u\n", x);
  printf ("Multibyte string character: %1s\n\n",pbuf);
  printf (" Convert when target is NULL\n");
  x = wcstombs( pbuf, pwcsEOL, MB_CUR_MAX);
  printf ("Character converted: %u\n",x);
  printf ("Multibyte string: %1s\n\n",pbuf);

}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.29.53 wctomb

Header File

stdlib.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
int wctomb(char *s, wchar_t wc);
```

Description

Converts **wchar_t** code to a multibyte character.

If s is not null, wctomb determines the number of bytes needed to represent the multibyte character corresponding to wc (including any change in shift state). The multibyte character is stored in s. At most MB_CUR_MAX characters are stored. If the value of wc is zero, wctomb is left in the initial state.

The behavior of wctomb is affected by the setting of LC_CTYPE category of the current locale.

Return Value

If s is a NULL pointer, wctomb returns a nonzero value if multibyte character encodings do have state-dependent encodings, and a zero value if they do not.

If s is not a NULL pointer, wctomb returns -1 if the wc value does not represent a valid multibyte character. Otherwise, wctomb returns the number of bytes that are contained in the multibyte character corresponding to wc. In no case will the return value be greater than the value of MB_CUR_MAX macro.

Example

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  int x;
  wchar_t wc = L'a';
  char *pmbNULL = NULL;
  char *pmb = (char *)malloc(sizeof( char ));
  printf (" Convert a wchar_t array into a multibyte string:\n");
  x = wctomb( pmb, wc);
  printf ("Character converted: %u\n", x);
  printf ("Multibyte string: %1s\n\n",pmb);
  printf (" Convert when target is NULL\n");
  x = wctomb( pmbNULL, wc);
  printf ("Character converted: %u\n",x);
  printf ("Multibyte stri ng: %1s\n\n",pmbNULL);

}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.30 string.h

The following functions, macros, and classes are provided in `string.h`:

Topics

Name	Description

_ismbblead, _ismbbtrail (see page 1152)	**Header File**
	mbstring.h
	Category
	Classification Routines
	Prototype
	`int _ismbblead(unsigned int c);`
	`int _ismbbtrail(unsigned int c);`
	Description
	_ismbblead and _ismbbtrail are used to test whether the argument c is the first or the second byte of a multibyte character.
	_ismbblead and _ismbbtrail are affected by the code page in use. You can set the code page by using the _setlocale function.
	Return Value
	If c is in the lead byte of a multibyte character, _ismbblead returns true.
	If c is in the trail byte of a multibyte character, _ismbbtrail returns a nonzero value.

_ismbclegal (see page 1153)	**Header File** mbstring.h **Category** Classification Routines **Prototype** `int _ismbclegal(unsigned int c);` **Description** _ismbclegal tests whether each byte of the c argument is in the code page that is currently in use. **Return Value** _ismbclegal returns a nonzero value if the argument c is a valid multibyte character on the current code page. Otherwise, the function returns zero.

_ismbslead, _ismbstrail (see page 1153)	**Header File**
	mbstring.h
	Category
	Classification Routines
	Prototype
	`int _ismbslead(const unsigned char *s1, const unsigned char *s2);`
	`int _ismbstrail(const unsigned char *s1, const unsigned char *s2);`
	Description
	The _ismbslead and _ismbstrail functions test the s1 argument to determine whether the s2 argument is a pointer to the lead byte or the trail byte. The test is case-sensitive.
	Return Value
	The _ismbslead and _ismbstrail routines return -1 if s2 points to a lead byte or a trail byte, respectively. If the test is false, the routines return zero.

_mbbtype (see page 1154)	**Header File** mbstring.h **Category** Classification Routines **Prototype** `int _mbbtype(unsigned char ch, int mode);` **Description** The _mbbtype function inspects the multibyte argument, character ch, to determine whether it is a single-byte character, or whether ch is the leadbyte or trailing byte in a multibyte character. The _mbbtype function can determine whether ch is an invalid character. **Return Value** The value that _mbbtype returns is one of the following manifest constants, defined in mbctype.h. The return value depends on the value of ch and the test which you want performed on ch.

_mbccpy (see page 1154)	**Header File** mbstring.h **Category** Memory and String Manipulation Routines **Prototype** `void _mbccpy(unsigned char *dest, unsigned char *src);` **Description** The _mbccpy function copies a multibyte character from src to dest. The _mbccpy function makes an implicit call to _ismbblead so that the src pointer references a lead byte. If src doesn't reference a lead byte, no copy is performed. **Return Value** None.
_mbsbtype (see page 1154)	**Header File** mbstring.h **Category** Classification Routines **Prototype** `int _mbsbtype(const unsigned char *str, size_t nbyte);` **Description** The nbyte argument specifies the number of bytes from the start of the zero-based string. The _mbsbtype function inspects the argument str to determine whether the byte at the position specified by nbyte is a single-byte character, or whether it is the leadbyte or trailing byte in a multibyte character. The _mbsbtype function can determine whether the byte pointed at is an invalid character or a NULL byte. Any invalid bytes in str before nbyte are ignored. **Return Value** The value that _mbsbtype returns is... more (see page 1154)
_mbsnbcmp (see page 1155)	**Header File** mbstring.h **Category** Memory and String Manipulation Routines **Prototype** `int _mbsnbcmp(const unsigned char *s1, const unsigned char s2, size_t maxlen);` **Description** _mbsnbcmp makes an case-sensitive comparison of s1 and s2 for no more than maxlen bytes. It starts with the first byte in each string and continues with subsequent bytes until the corresponding bytes differ or until it has examined maxlen bytes. _mbsnbcmp is case sensitive. _mbsnbcmp is not affected by locale. _mbsnbcmp compares bytes based on the current multibyte code page. **Return Value** • _mbsnbcmp returns an integer value based on the result of comparing s1 (or part of... more (see page 1155)

_mbsnbcnt, _mbsnccnt, _strncnt, _wcsncnt (see page 1155)	**Header File**
	mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`size_t _mbsnbcnt(const unsigned char * str, size_t nmbc);`
	`size_t _mbsnccnt(const unsigned char * str, size_t nbyte);`
	Description
	If _MBCS is defined:
	_mbsnbcnt is mapped to the portable macro _tcsnbcnt
	_mbsnccnt is mapped to the portable macro _tcsnccnt If _UNICODE is defined:
	both _mbsnbcnt and _mbsnccnt are mapped to the _wcsncnt macro
	If neither _MBCS nor _UNICODE are defined.
	_tcsnbcnt and _tcsnccnt are mapped to the _strncnt macro _strncnt is the single-byte version of these functions. _wcsncnt is the wide-character version of these functions.
	_strncnt and _wcsncnt are available only for... more (see page 1155)

_mbsnbcoll, _mbsnbicoll (see page 1156)	**Header File**
	mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int _mbsnbcoll(const unsigned char *s1, const unsigned char *s2, maxlen);`
	`int _mbsnbicoll(const unsigned char *s1, const unsigned char *s2, maxlen);`
	Description
	_mbsnbicoll is the case-insensitive version of _mbsnbcoll.
	These functions collate the strings specified by arguments s1 and s2. The collation order is determined by lexicographic order as specified by the current multibyte code page. At most, maxlen number of bytes are collated.
	Note: Note:
	The lexicographic order is not always the same as the order of characters in the character set.If the last byte in s1 or s2 is... more (see page 1156)

_mbsnbcpy (see page 1157)	**Header File** mbstring.h **Category** Memory and String Manipulation Routines **Prototype** `unsigned char *_mbsnbcpy(unsigned char *dest, unsigned char *src, size_t maxlen);` **Description** The _mbsnbcpy function copies at most maxlen number of characters from the src buffer to the dest buffer. The dest buffer is null terminated after the copy. It is the user's responsibility to be sure that dest is large enough to allow the copy. An improper buffer size can result in memory corruption. **Return Value** The function returns dest.
_mbsnbicmp (see page 1157)	**Header File** mbstring.h **Category** Memory and String Manipulation Routines **Prototype** `int _mbsnbicmp(const unsigned char *s1, const unsigned char s2, size_t maxlen);` **Description** _mbsnbicmp ignores case while making a comparison of s1 and s2 for no more than maxlen bytes. It starts with the first byte in each string and continues with subsequent bytes until the corresponding bytes differ or until it has examined maxlen bytes. _mbsnbicmp is not case sensitive. _mbsnbicmp is not affected by locale. _mbsnbicmp compares bytes based on the current multibyte code page. **Return Value** • _mbsnbicmp returns an integer value based on the result of comparing s1... more (see page 1157)
_mbsnbset (see page 1158)	**Header File** mbstring.h **Category** Memory and String Manipulation Routines **Prototype** `unsigned char *_mbsnbset(unsigned char str, unsigned int ch, size_t maxlen);` **Description** _mbsnbset sets at most maxlen number of bytes in the string str to the character ch. The argument ch can be a single or multibyte character. The function quits if the terminating null character is found before maxlen is reached. If ch is a multibyte character that cannot be accomodated at the end of str, the last character in str is set to a blank character. **Return Value** strset returns str.

_mbsninc, _strninc, _wcsninc (see page 1158)	**Header File** mbstring.h **Category** Memory and String Manipulation Routines **Prototype** `unsigned char *_mbsninc(const unsigned char *str, size_t num);` **Description** The functions increment the character array str by num number of characters. These functions should be accessed through the portable macro, _tcsninc, defined in tchar.h. **Return value** The functions return a pointer to the resized character string specified by the argument str.
_mbsspnp, _strspnp, _wcsspnp (see page 1158)	**Header File** mbstring.h **Category** Memory and String Manipulation Routines **Prototype** `unsigned char *_mbsspnp(const unsigned char *s1, const unsigned char *s2);` **Description** Each of these functions search for the first character in s1 that is not contained in s2. Use the portable macro, _tcsspnp, defined in tchar.h, to access these functions. **Return Value** The functions return a pointer to the first character in s1 that is not found in the character set for s2. If every character from s1 is found in s2, each of the functions return NULL.

_strdec, mbsdec, _wcsdec (see page 1159)	**Header File**
	mbstring.h, tchar.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`unsigned char *_mbsdec(const unsigned char *s, const unsigned char *p);`
	`inline char *_strdec(const char * s1, const char * s2) { return (char *)(s1,(s2-1));`
	`// From tchar.h`
	`#define _tcsdec _strdec`
	`#define _tcsdec _wcsdec`
	Description
	_mbsdec returns a pointer p indicating the character in string s back to 1 byte backward. If there are no more characters before string p (it is the same position as s), _mbsdec returns a **null** pointer.
	_strdec is the single-byte version of this function.
	_wcsdec is the wide-character version of this function.
	Return Value... more (see page 1159)

_strerror (see page 1159)	**Header File** string.h, stdio.h **Category** Memory and String Manipulation Routines **Prototype** `char *_strerror(const char *s);` **Description** Builds a customized error message. _strerror lets you generate customized error messages; it returns a pointer to a null-terminated string containing an error message. • If s is null, the return value points to the most recent error message. • If s is not null, the return value contains s (your customized error message), a colon, a space, the most-recently generated system error message, and a new line. s should be 94 characters or less. **Return Value** _strerror returns a pointer to a constructed error string.... more (see page 1159)

_strinc, mbsinc, _wcsinc (see page 1160)	**Header File**
	mbstring.h, tchar.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`unsigned char *_mbsinc(const unsigned char *p);`
	`// From tchar.h`
	`#define _tcsinc _strinc`
	`#define _tcsinc _wcsinc`
	`inline char * strinc(const char * s) { return (char *)(s+1); }`
	Description
	_mbsinc increments a string pointer by one byte.
	_strdec is the single-byte version of this function.
	_wcsdec is the wide-character version of this function.
	Return Value
	Returns a pointer that is forwarded by 1 byte.

_strnextc,_mbsnextc,_wcsnextc (see page 1161)	**Header File**
	tchar.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`unsigned int _strnextc(const char *str);`
	`unsigned int _mbsnextc (const unsigned char *str);`
	Description
	These routines should be accessed by using the portable _tcsnextc function. The functions inspect the current character in str. The pointer to str is not advanced.
	Return Value
	The functions return the integer value of the character pointed to by str.
	Example

stpcpy, _wstpcpy, _stpcpy (see page 1161)	**Header File**
	string.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`char *stpcpy(char *dest, const char *src);`
	`wchar * _wcspcpy(wchar *dest, const wchar *src);`
	Description
	Copies one string into another.
	_stpcpy copies the string src to dest, stopping after the terminating null character of src has been reached.
	Return Value
	stpcpy returns a pointer to the terminating null character of dest.
	If UNICODE is defined, _wcspcpy returns a pointer to the terminating null character of the **wchar_t** dest string.
	Example

strcat, _mbscat, wcscat (see page 1162)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`char *strcat(char *dest, const char *src);`
	`wchar_t *wcscat(wchar_t *dest, const wchar_t *src);`
	`unsigned char *_mbscat(unsigned char *dest, const unsigned char *src);`
	Description
	Appends one string to another.
	strcat appends a copy of src to the end of dest. The length of the resulting string is strlen(dest) + strlen(src).
	Return Value
	strcat returns a pointer to the concatenated strings.
	Example

strchr, _mbschr, wcschr (see page 1163)	**Header File**
	string.h
	Category
	Memory and String Manipulation Routines, Inline Routines, C++ Prototyped Routines
	Prototype
	`char *strchr(const char *s, int c);/* C only */`
	`const char *strchr(const char *s, int c);// C++ only`
	`char *strchr( char *s, int c);// C++ only`
	`wchar_t *wcschr(const wchar_t *s, int c);`
	`unsigned char * _mbschr(const unsigned char *s, unsigned int c);`
	Description
	Scans a string for the first occurrence of a given character.
	strchr scans a string in the forward direction, looking for a specific character. strchr finds the first occurrence of the character c in the string s. The null-terminator is considered to... more (see page 1163)

strcmp, _mbscmp, wcscmp (see page 1164)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`int strcmp(const char *s1, const char *s2);`
	`int wcscmp(const wchar_t *s1, const wchar_t *s2);`
	`int _mbscmp(const unsigned char *s1, const unsigned char *s2);`
	Description
	Compares one string to another.
	strcmp performs an unsigned comparison of s1 to s2, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.
	Return Value

strcmpi (see page 1165)	**Header File**
	string.h, wchar.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int strcmpi(const char *s1, const char *s2);`
	`int _wcscmpi(const wchar_t *s1, const wchar_t *s2);`
	Description
	Compares one string to another, without case sensitivity.
	strcmpi performs an unsigned comparison of s1 to s2, without case sensitivity (same as stricmp--implemented as a macro).
	It returns a value (< 0, 0, or > 0) based on the result of comparing s1 (or part of it) to s2 (or part of it).
	The routine strcmpi is the same as stricmp. strcmpi is implemented through a macro in string.h and translates calls from strcmpi... more (see page 1165)

strcoll,_stricoll, _mbscoll, _mbsicoll, wcscoll, _wcsicoll (see page 1166)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int strcoll(const char *s1, const char *s2);`
	`int wcscoll(const wchar_t *s1, const wchar_t *s2);`
	`int _stricoll(const char *s1, const char *s2);`
	`int _wcsicoll(const wchar_t *s1, wconst_t char *s2);`
	`int _mbscoll(const unsigned char *s1, const unsigned char *s2);`
	`int _mbsicoll(const unsigned char *s1, const unsigned char *s2);`
	Description
	Compares two strings.
	strcoll compares the string pointed to by s1 to the string pointed to by s2, according to the current locale's LC_COLLATE category.
	_stricoll performs like strcoll but is not case sensitive.
	Note: Note
	_stricoll does not compare string according... more (see page 1166)

strcpy (see page 1167)	**Header File**
	string.h, wchar.h, mbstring.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`char *strcpy(char *dest, const char *src);`
	`wchar_t *wcscpy(wchar_t *dest, const wchar_t *src);`
	`unsigned char *_mbscpy(unsigned char *dest, const unsigned char *src);`
	Description
	Copies one string into another.
	Copies string src to dest, stopping after the terminating null character has been moved.
	Return Value
	strcpy returns dest.
	Example

strcspn, _mbscspn, wcscspn (see page 1168)	**Header File**
	string.h, wchar.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`size_t strcspn(const char *s1, const char *s2);`
	`size_t wcscspn(const wchar_t *s1, const wchar_t *s2);`
	`size_t _mbscspn(const unsigned char *s1, const unsigned char *s2);`
	Description
	Scans a string for the initial segment not containing any subset of a given set of characters.
	The strcspn functions search s1 until any one of the characters contained in s2 is found. The number of characters which were read in s1 is the return value. The string termination character is not counted. Neither string is altered during the search.
	Return Value
	strcspn returns... more (see page 1168)

strdup, _mbsdup, _wcsdup (see page 1169)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`char *strdup(const char *s);`
	`wchar_t *_wcsdup(const wchar_t *s);`
	`unsigned char *_mbsdup(const wchar_t *s);`
	Description
	Copies a string into a newly created location.
	strdup makes a duplicate of string s, obtaining space with a call to malloc. The allocated space is (strlen(s) + 1) bytes long. The user is responsible for freeing the space allocated by strdup when it is no longer needed.
	Return Value
	strdup returns a pointer to the storage location containing the duplicated string, or returns null if space could not be allocated.
	Example

strerror (see page 1170)	**Header File** string.h **Category** Memory and String Manipulation Routines **Prototype** `char *strerror(int errnum);` **Description** Returns a pointer to an error message string. strerror takes an **int** parameter errnum, an error number, and returns a pointer to an error message string associated with errnum. **Return Value** strerror returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to strerror. **Example**

stricmp, _mbsicmp, _wcsicmp (see page 1170)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int stricmp(const char *s1, const char *s2);`
	`int _wcsicmp(const wchar_t *s1, const wchar_t *s2);`
	`int _mbsicmp(const unsigned char *s1, const unsigned char *s2);`
	Description
	Compares one string to another, without case sensitivity.
	stricmp performs an unsigned comparison of s1 to s2, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.
	It returns a value (< 0, 0, or > 0) based on the result of... more (see page 1170)

strlwr, _mbslwr, _wcslwr (see page 1171)	**Header File**
	string.h, mbstring.h
	Category
	Conversion Routines, Memory and String Manipulation Routines
	Prototype
	`char *strlwr(char *s);`
	`wchar_t *_wcslwr(wchar_t *s);`
	`unsigned char *_mbslwr(unsigned char *s);`
	Description
	Converts uppercase letters in a string to lowercase.
	strlwr converts uppercase letters in string s to lowercase according to the current locale's LC_CTYPE category. For the C locale, the conversion is from uppercase letters (A to Z) to lowercase letters (a to z). No other characters are changed.
	Return Value
	strlwr returns a pointer to the string s.
	Example

strncat (see page 1172)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`char *strncat(char *dest, const char *src, size_t maxlen);`
	`wchar_t *wcsncat(wchar_t *dest, const wchar_t *src, size_t maxlen);`
	`unsigned char *_mbsncat(unsigned char *dest, const unsigned char *src, size_t maxlen);`
	`unsigned char *_mbsnbcat(unsigned char *__dest, const unsigned char *__src, _SIZE_T __maxlen);`
	Description
	Appends a portion of one string to another.
	strncat copies at most maxlen characters of src to the end of dest and then appends a null character. The maximum length of the resulting string is strlen(dest) + maxlen.
	For _mbsnbcat, if the second byte of 2-bytes character is... more (see page 1172)

strncmp, _mbsncmp, wcsncmp (see page 1173)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`int strncmp(const char *s1, const char *s2, size_t maxlen);`
	`int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t maxlen);`
	`int _mbsncmp(const unsigned char *s1, const unsigned char *s2, size_t maxlen);`
	`#define _mbccmp(__s1, __s2) _mbsncmp((__s1),(__s2),1)`
	Description
	Compares a portion of one string to a portion of another.
	strncmp makes the same unsigned comparison as strcmp, but looks at no more than maxlen characters. It starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until it has examined maxlen characters.... more (see page 1173)

strncmpi, wcsncmpi (see page 1174)	**Header File**
	string.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int strncmpi(const char *s1, const char *s2, size_t n);`
	`int wcsncmpi(const wchar_t *s1, const wchar_t *s2, size_t n);`
	Description
	Compares a portion of one string to a portion of another, without case sensitivity.
	strncmpi performs a signed comparison of s1 to s2, for a maximum length of n bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until n characters have been examined. The comparison is not case sensitive. (strncmpi is the same as strnicmp--implemented as a macro). It... more (see page 1174)

strncoll, strnicoll, _mbsncoll, _mbsnicoll, _wcsncoll, _wcsnicoll (see page 1175)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int _strncoll(const char *s1, const char *s2, size_t n);`
	`int _wcsncoll(const wchar_t *s1, const wchar_t *s2, size_t n);`
	`int _strnicoll(const char *s1, const char *s2, size_t n);`
	`int _wcsnicoll(const wchar_t *s1, const wchar_t *s2, size_t n);`
	`int _mbsncoll(const unsigned char *s1, const unsigned char *s2, size_t n);`
	`int _mbsnicoll(const unsigned char *s1, const unsigned char *s2, size_t n);`
	Description
	_strncoll compares n number of elements from the string pointed to by s1 to the string pointed to by s2, according to the current locale's LC_COLLATE category.
	_strnicoll performs like... more (see page 1175)

strncpy, _mbsncpy, wcsncpy (see page 1176)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`char *strncpy(char *dest, const char *src, size_t maxlen);`
	`wchar_t *wcsncpy(wchar_t *dest, const wchar_t *src, size_t maxlen);`
	`unsigned char *_mbsncpy(unsigned char *dest, const unsigned char *src, size_t maxlen);`
	Description
	Copies a given number of bytes from one string into another, truncating or padding as necessary.
	strncpy copies up to maxlen characters from src into dest, truncating or null-padding dest. The target string, dest, might not be null-terminated if the length of src is maxlen or more.
	Return Value
	strncpy returns dest.
	Example

strnicmp, _mbsnicmp, _wcsnicmp (see page 1176)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`int strnicmp(const char *s1, const char *s2, size_t maxlen);`
	`int _wcsnicmp(const wchar_t *s1, const wchar_t *s2, size_t maxlen);`
	`int _mbsnicmp(const unsigned char *s1, const unsigned char *s2, size_t maxlen);`
	Description
	Compares a portion of one string to a portion of another, without case sensitivity.
	strnicmp performs a signed comparison of s1 to s2, for a maximum length of maxlen bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison... more (see page 1176)

strnset, _mbsnset, _wcsnset (see page 1177)	**Header File**
	string.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`char *strnset(char *s, int ch, size_t n);`
	`wchar_t *_wcsnset(wchar_t *s, wchar_t ch, size_t n);`
	`unsigned char *_mbsnset(unsigned char *s, unsigned int ch, size_t n);`
	Description
	Sets a specified number of characters in a string to a given character.
	strnset copies the character ch into the first n bytes of the string s. If n > strlen(s), then strlen(s) replaces n. It stops when n characters have been set, or when a null character is found.
	Return Value
	Each of these functions return s.
	Example

strpbrk, _mbspbrk, wcspbrk (see page 1178)	**Header File**
	string.h, mbstring.h
	Category
	C++ Prototyped Routines, Memory and String Manipulation Routines
	Prototype
	`char *strpbrk(const char *s1, const char *s2); /* C only */`
	`const char *strpbrk(const char *s1, const char *s2); // C++ only`
	`char *strpbrk(char *s1, const char *s2); // C++ only`
	`wchar_t * wcspbrk(const wchar_t *s1, const wchar_t *s2);`
	`unsigned char *_mbspbrk(const unsigned char *s1, const unsigned char *s2);`
	Description
	Scans a string for the first occurrence of any character from a given set.
	strpbrk scans a string, s1, for the first occurrence of any character appearing in s2.
	Return Value
	strpbrk returns a pointer to... more (see page 1178)

strrchr, _mbsrchr, wcsrchr (see page 1179)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines, Inline Routines, C++ Prototyped Routines
	Prototype
	`char *strrchr(const char *s, int c); /* C only */`
	`const char *strrchr(const char *s, int c); // C++ only`
	`char *strrchr(char *s, int c); // C++ only`
	`wchar_t *wcsrchr(const wchar_t *s, wchar_t c);`
	`unsigned char * _mbsrchr(const unsigned char *s, unsigned int c);`
	Description
	Scans a string for the last occurrence of a given character.
	strrchr scans a string in the reverse direction, looking for a specific character. strrchr finds the last occurrence of the character c in the string s. The null-terminator... more (see page 1179)

strrev, _mbsrev, _wcsrev (see page 1180)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`char *strrev(char *s);`
	`wchar_t *_wcsrev(wchar_t *s);`
	`unsigned char *_mbsrev(unsigned char *s);`
	Description
	Reverses a string.
	strrev changes all characters in a string to reverse order, except the terminating null character. (For example, it would change string\0 to gnirts\0.)
	Return Value
	strrev returns a pointer to the reversed string.
	Example

strset, _mbsset, _wcsset (see page 1181)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines, Inline Routines
	Prototype
	`char *strset(char *s, int ch);`
	`wchar_t *_wcsset(wchar_t *s, wchar_t ch);`
	`unsigned char *_mbsset(unsigned char *s, unsigned int ch);`
	Description
	Sets all characters in a string to a given character.
	strset sets all characters in the string s to the character ch. It quits when the terminating null character is found.
	Return Value
	strset returns s.
	Example

strspn, _mbsspn, wcsspn (see page 1182)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`size_t strspn(const char *s1, const char *s2);`
	`size_t wcsspn(const wchar_t *s1, const wchar_t *s2);`
	`size_t _mbsspn(const unsigned char *s1, const unsigned char *s2);`
	Description
	Scans a string for the first segment that is a subset of a given set of characters.
	strspn finds the initial segment of string s1 that consists entirely of characters from string s2.
	Return Value
	strspn returns the length of the initial segment of s1 that consists entirely of characters from s2.
	Example

strstr, _mbsstr, wcsstr (see page 1182)	**Header File**
	string.h
	Category
	C++ Prototyped Routines, Memory and String Manipulation Routines
	Prototype
	`char *strstr(const char *s1, const char *s2); /* C only */`
	`const char *strstr(const char *s1, const char *s2); // C++ only`
	`char *strstr(char *s1, const char *s2); // C++ only`
	`wchar_t * wcsstr(const wchar_t *s1, const wchar_t *s2);`
	`unsigned char * _mbsstr(const unsigned char *s1, const unsigned char *s2);`
	Description
	Scans a string for the occurrence of a given substring.
	strstr scans s1 for the first occurrence of the substring s2.
	Return Value
	strstr returns a pointer to the element in s1, where s2 begins (points... more (see page 1182)

strtok, _mbstok, wcstok (see page 1183)	**Header File**
	string.h, mbstring.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`char *strtok(char *s1, const char *s2);`
	`wchar_t *wcstok(wchar_t *s1, const wchar_t *s2);`
	`unsigned char *_mbstok(unsigned char *s1, const unsigned char *s2);`
	Description
	Searches one string for tokens, which are separated by delimiters defined in a second string.
	strtok considers the string s1 to consist of a sequence of zero or more text tokens, separated by spans of one or more characters from the separator string s2.
	The first call to strtok returns a pointer to the first character of the first token in s1 and writes a null character... more (see page 1183)

strupr, _mbsupr, _wcsupr (see page 1184)	**Header File**
	string.h, mbstring.h
	Category
	Conversion Routines, Memory and String Manipulation Routines
	Prototype
	`char *strupr(char *s);`
	`wchar_t *_wcsupr(wchar_t *s);`
	`unsigned char *_mbsupr(unsigned char *s);`
	Description
	Converts lowercase letters in a string to uppercase.
	strupr converts lowercase letters in string s to uppercase according to the current locale's LC_CTYPE category. For the default C locale, the conversion is from lowercase letters (a to z) to uppercase letters (A to Z). No other characters are changed.
	Return Value
	strupr returns s.
	Example

strxfrm, wcsxfrm (see page 1185)	**Header File**
	string.h
	Category
	Memory and String Manipulation Routines
	Prototype
	`size_t strxfrm(char *target, const char *source, size_t n);`
	`size_t wcsxfrm(wchar_t *target, const wchar_t *source, size_t n);`
	Description
	Transforms a portion of a string to a specified collation.
	strxfrm transforms the string pointed to by source into the string target for no more than n characters. The transformation is such that if the strcmp function is applied to the resulting strings, its return corresponds with the return values of the strcoll function.
	No more than n characters, including the terminating null character, are copied to target.
	strxfrm transforms a character string... more (see page 1185)

3.1.4.30.1 _ismbblead, _ismbbtrail

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _ismbblead(unsigned int c);
```

```
int _ismbbtrail(unsigned int c);
```

Description

_ismbblead and _ismbbtrail are used to test whether the argument c is the first or the second byte of a multibyte character.

_ismbblead and _ismbbtrail are affected by the code page in use. You can set the code page by using the _setlocale function.

Return Value

If c is in the lead byte of a multibyte character, _ismbblead returns true.

If c is in the trail byte of a multibyte character, _ismbbtrail returns a nonzero value.

3.1.4.30.2 _ismbclegal

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _ismbclegal(unsigned int c);
```

Description

_ismbclegal tests whether each byte of the c argument is in the code page that is currently in use.

Return Value

_ismbclegal returns a nonzero value if the argument c is a valid multibyte character on the current code page. Otherwise, the function returns zero.

3.1.4.30.3 _ismbslead, _ismbstrail

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _ismbslead(const unsigned char *s1, const unsigned char *s2);
int _ismbstrail(const unsigned char *s1, const unsigned char *s2);
```

Description

The _ismbslead and _ismbstrail functions test the s1 argument to determine whether the s2 argument is a pointer to the lead byte or the trail byte. The test is case-sensitive.

Return Value

The _ismbslead and _ismbstrail routines return -1 if s2 points to a lead byte or a trail byte, respectively. If the test is false, the routines return zero.

3.1.4.30.4 _mbbtype

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _mbbtype(unsigned char ch, int mode);
```

Description

The _mbbtype function inspects the multibyte argument, character ch, to determine whether it is a single-byte character, or whether ch is the leadbyte or trailing byte in a multibyte character. The _mbbtype function can determine whether ch is an invalid character.

Return Value

The value that _mbbtype returns is one of the following manifest constants, defined in mbctype.h. The return value depends on the value of ch and the test which you want performed on ch.

3.1.4.30.5 _mbccpy

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
void _mbccpy(unsigned char *dest, unsigned char *src);
```

Description

The _mbccpy function copies a multibyte character from src to dest. The _mbccpy function makes an implicit call to _ismbblead so that the src pointer references a lead byte. If src doesn't reference a lead byte, no copy is performed.

Return Value

None.

3.1.4.30.6 _mbsbtype

Header File

mbstring.h

Category

Classification Routines

Prototype

```
int _mbsbtype(const unsigned char *str, size_t nbyte);
```

Description

The nbyte argument specifies the number of bytes from the start of the zero-based string.

The _mbsbtype function inspects the argument str to determine whether the byte at the position specified by nbyte is a single-byte character, or whether it is the leadbyte or trailing byte in a multibyte character. The _mbsbtype function can determine whether the byte pointed at is an invalid character or a NULL byte.

Any invalid bytes in str before nbyte are ignored.

Return Value

The value that _mbsbtype returns is one of the following manifest constants, defined in mbctype.h.

3.1.4.30.7 _mbsnbcmp

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int _mbsnbcmp(const unsigned char *s1, const unsigned char s2, size_t maxlen);
```

Description

_mbsnbcmp makes an case-sensitive comparison of s1 and s2 for no more than maxlen bytes. It starts with the first byte in each string and continues with subsequent bytes until the corresponding bytes differ or until it has examined maxlen bytes.

_mbsnbcmp is case sensitive.

_mbsnbcmp is not affected by locale.

_mbsnbcmp compares bytes based on the current multibyte code page.

Return Value

- _mbsnbcmp returns an integer value based on the result of comparing s1 (or part of it) to s2 (or part of it):
- < 0 if s1 is less than s2
- == 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

3.1.4.30.8 _mbsnbcnt, _mbsnccnt, _strncnt, _wcsncnt

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
size_t _mbsnbcnt(const unsigned char * str, size_t nmbc);
size_t _mbsnccnt(const unsigned char * str, size_t nbyte);
```

Description

If _MBCS is defined:

_mbsnbcnt is mapped to the portable macro _tcsnbcnt

_mbsnccnt is mapped to the portable macro _tcsnccnt If _UNICODE is defined:

both _mbsnbcnt and _mbsnccnt are mapped to the _wcsncnt macro

If neither _MBCS nor _UNICODE are defined.

_tcsnbcnt and _tcsnccnt are mapped to the _strncnt macro _strncnt is the single-byte version of these functions. _wcsncnt is the wide-character version of these functions.

_strncnt and _wcsncnt are available only for generic-text mappings. They should not be used directly.

_mbsnbcnt examines the first nmbc multibyte characters of the str argument. The function returns the number of bytes found in the those characters.

_mbsnccnt examines the first nmbc bytes of the str argument. The function returns the number of characters found in those bytes. If NULL is encountered in the second byte of a multibyte character, the whole character is considered NULL and will not be included in the return value.

Each of the functions ends its examination of the str argument if NULL is reached before the specified number of characters or bytes is examined.

If str has fewer than the specified number of characters or bytes, the function return the number of characters or bytes found in str.

Return Value

_mbsnbcnt returns the number of bytes found.

_mbsnccnt returns the number of characters found.

If nmbc or nbyte are less than zero, the functions return 0.

3.1.4.30.9 _mbsnbcoll, _mbsnbicoll

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int _mbsnbcoll(const unsigned char *s1, const unsigned char *s2, maxlen);
int _mbsnbicoll(const unsigned char *s1, const unsigned char *s2, maxlen);
```

Description

_mbsnbicoll is the case-insensitive version of _mbsnbcoll.

These functions collate the strings specified by arguments s1 and s2. The collation order is determined by lexicographic order as specified by the current multibyte code page. At most, maxlen number of bytes are collated.

Note: Note:

The lexicographic order is not always the same as the order of characters in the character set.If the last byte in s1 or s2 is a

leadbyte, it is not compared.

Return Value

- Each of these functions return an integer value based on the result of comparing s1 (or part of it) to s2 (or part of it):
- < 0 if s1 is less than s2
- == 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

On error, each of these functions returns _**NLSCMPERROR**.

3.1.4.30.10 _mbsnbcpy

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsnbcpy(unsigned char *dest, unsigned char *src, size_t maxlen);
```

Description

The _mbsnbcpy function copies at most maxlen number of characters from the src buffer to the dest buffer. The dest buffer is null terminated after the copy.

It is the user's responsibility to be sure that dest is large enough to allow the copy. An improper buffer size can result in memory corruption.

Return Value

The function returns dest.

3.1.4.30.11 _mbsnbicmp

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int _mbsnbicmp(const unsigned char *s1, const unsigned char s2, size_t maxlen);
```

Description

_mbsnbicmp ignores case while making a comparison of s1 and s2 for no more than maxlen bytes. It starts with the first byte in each string and continues with subsequent bytes until the corresponding bytes differ or until it has examined maxlen bytes.

_mbsnbicmp is not case sensitive.

_mbsnbicmp is not affected by locale.

_mbsnbicmp compares bytes based on the current multibyte code page.

Return Value

- _mbsnbicmp returns an integer value based on the result of comparing s1 (or part of it) to s2 (or part of it):
- < 0 if s1 is less than s2
- == 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

3.1.4.30.12 _mbsnbset

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsnbset(unsigned char str, unsigned int ch, size_t maxlen);
```

Description

_mbsnbset sets at most maxlen number of bytes in the string str to the character ch. The argument ch can be a single or multibyte character.

The function quits if the terminating null character is found before maxlen is reached. If ch is a multibyte character that cannot be accomodated at the end of str, the last character in str is set to a blank character.

Return Value

strset returns str.

3.1.4.30.13 _mbsninc, _strninc, _wcsninc

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsninc(const unsigned char *str, size_t num);
```

Description

The functions increment the character array str by num number of characters.

These functions should be accessed through the portable macro, _tcsninc, defined in tchar.h.

Return value

The functions return a pointer to the resized character string specified by the argument str.

3.1.4.30.14 _mbsspnp, _strspnp, _wcsspnp

Header File

mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsspnp(const unsigned char *s1, const unsigned char *s2);
```

Description

Each of these functions search for the first character in s1 that is not contained in s2.

Use the portable macro, _tcsspnp, defined in tchar.h, to access these functions.

Return Value

The functions return a pointer to the first character in s1 that is not found in the character set for s2.

If every character from s1 is found in s2, each of the functions return NULL.

3.1.4.30.15 _strdec, mbsdec, _wcsdec

Header File

mbstring.h, tchar.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsdec(const unsigned char *s, const unsigned char *p);
inline char *_strdec(const char * s1, const char * s2) { return (char *)(s1,(s2-1));
// From tchar.h
#define _tcsdec _strdec
#define _tcsdec _wcsdec
```

Description

_mbsdec returns a pointer p indicating the character in string s back to 1 byte backward. If there are no more characters before string p (it is the same position as s), _mbsdec returns a **null** pointer.

_strdec is the single-byte version of this function.

_wcsdec is the wide-character version of this function.

Return Value

Returns a pointer back to 1 byte, or **null** pointer if there is no character before p.

3.1.4.30.16 _strerror

Header File

string.h, stdio.h

Category

Memory and String Manipulation Routines

Prototype

```
char *_strerror(const char *s);
```

Description

Builds a customized error message.

_strerror lets you generate customized error messages; it returns a pointer to a null-terminated string containing an error message.

- If s is null, the return value points to the most recent error message.
- If s is not null, the return value contains s (your customized error message), a colon, a space, the most-recently generated system error message, and a new line. s should be 94 characters or less.

Return Value

_strerror returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to _strerror.

Example

```
#include <stdio.h>
#include <errno.h>
int main(void)
{
   char *buffer;
   buffer = strerror(errno);
   printf("Error: %s\n", buffer);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.30.17 _strinc, mbsinc, _wcsinc

Header File

mbstring.h, tchar.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned char *_mbsinc(const unsigned char *p);

// From tchar.h

#define _tcsinc _strinc

#define _tcsinc _wcsinc

inline char * strinc(const char * s) { return (char *)(s+1); }
```

Description

_mbsinc increments a string pointer by one byte.

_strdec is the single-byte version of this function.

_wcsdec is the wide-character version of this function.

Return Value

Returns a pointer that is forwarded by 1 byte.

3.1.4.30.18 _strnextc,_mbsnextc,_wcsnextc

Header File

tchar.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
unsigned int _strnextc(const char *str);
```

```
unsigned int _mbsnextc (const unsigned char *str);
```

Description

These routines should be accessed by using the portable _tcsnextc function. The functions inspect the current character in str. The pointer to str is not advanced.

Return Value

The functions return the integer value of the character pointed to by str.

Example

```
#include <tchar.h>
#include <stdio.h>
int main()
{
   unsigned int retval = 0;
   const unsigned char *string = "ABC";
   retval = _strnextc(string);
   printf("The starting character:%c", retval);
   retval = _strnextc(++string);
   printf("\nThe next character:%c", retval);
   return 0;
}
/***
The starting character:A
The next character:B
***/
```

3.1.4.30.19 stpcpy, _wstpcpy, _stpcpy

Header File

string.h

Category

Memory and String Manipulation Routines

Prototype

```
char *stpcpy(char *dest, const char *src);
```

```
wchar * _wcspcpy(wchar *dest, const wchar *src);
```

Description

Copies one string into another.

_stpcpy copies the string src to dest, stopping after the terminating null character of src has been reached.

Return Value

stpcpy returns a pointer to the terminating null character of dest.

If UNICODE is defined, _wcspcpy returns a pointer to the terminating null character of the **wchar_t** dest string.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char string[10];
   char *str1 = "abcdefghi";
   stpcpy(string, str1);
   printf("%s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_stpcpy		+		
_wcspcpy		+		

3.1.4.30.20 strcat, _mbscat, wcscat

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strcat(char *dest, const char *src);
wchar_t *wcscat(wchar_t *dest, const wchar_t *src);
unsigned char *_mbscat(unsigned char *dest, const unsigned char *src);
```

Description

Appends one string to another.

strcat appends a copy of src to the end of dest. The length of the resulting string is strlen(dest) + strlen(src).

Return Value

strcat returns a pointer to the concatenated strings.

Example

```
#include <string.h>
#include <stdio.h>
```

```
int main(void)
{
   char destination[25];
   char *blank = " ", *c = "C++", *CodeGear = "CodeGear";
   strcpy(destination, CodeGear);
   strcat(destination, blank);
   strcat(destination, c);
   printf("%s\n", destination);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcat	+	+	+	+
_mbscat		+		
wcscat		+	+	+

3.1.4.30.21 strchr, _mbschr, wcschr

Header File

string.h

Category

Memory and String Manipulation Routines, Inline Routines, C++ Prototyped Routines

Prototype

```
char *strchr(const char *s, int c);/* C only */
```

```
const char *strchr(const char *s, int c);// C++ only
```

```
char *strchr( char *s, int c);// C++ only
```

```
wchar_t *wcschr(const wchar_t *s, int c);
```

```
unsigned char * _mbschr(const unsigned char *s, unsigned int c);
```

Description

Scans a string for the first occurrence of a given character.

strchr scans a string in the forward direction, looking for a specific character. strchr finds the first occurrence of the character c in the string s. The null-terminator is considered to be part of the string.

For example:

```
strchr(strs,0)
```

returns a pointer to the terminating null character of the string strs.

Return Value

strchr returns a pointer to the first occurrence of the character c in s; if c does not occur in s, strchr returns null.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char string[15];
```

```
    char *ptr, c = 'r';
    strcpy(string, "This is a string");
    ptr = strchr(string, c);
    if (ptr)
       printf("The character %c is at position: %d\n", c, ptr-string);
    else
       printf("The character was not found\n");
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strchr	+	+	+	+
_mbschr		+		
wcschr		+	+	+

3.1.4.30.22 strcmp, _mbscmp, wcscmp

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
int strcmp(const char *s1, const char *s2);
```

```
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

```
int _mbscmp(const unsigned char *s1, const unsigned char *s2);
```

Description

Compares one string to another.

strcmp performs an unsigned comparison of s1 to s2, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
char *buf1 = "aaa", *buf2 = "bbb", *buf3 = "ccc";
int ptr;
ptr = strcmp(buf2, buf1);
if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");
```

```
   ptr = strcmp(buf2, buf3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcmp	+	+	+	+
_mbscmp		+		
wcscmp		+	+	+

3.1.4.30.23 strcmpi

Header File

string.h, wchar.h

Category

Memory and String Manipulation Routines

Prototype

```
int strcmpi(const char *s1, const char *s2);
```

```
int _wcscmpi(const wchar_t *s1, const wchar_t *s2);
```

Description

Compares one string to another, without case sensitivity.

strcmpi performs an unsigned comparison of s1 to s2, without case sensitivity (same as stricmp--implemented as a macro).

It returns a value (< 0, 0, or > 0) based on the result of comparing s1 (or part of it) to s2 (or part of it).

The routine strcmpi is the same as stricmp. strcmpi is implemented through a macro in string.h and translates calls from strcmpi to stricmp. Therefore, in order to use strcmpi, you must include the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
/* strncmpi example */
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *buf1 = "BBB", *buf2 = "bbb";
   int ptr;
   ptr = strcmpi(buf2, buf1);
   if (ptr > 0)
```

```
      printf("buffer 2 is greater than buffer 1\n");
   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");
   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcmpi		+		
_wcscmpi		+		

3.1.4.30.24 strcoll,_stricoll, _mbscoll, _mbsicoll, wcscoll, _wcsicoll

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int strcoll(const char *s1, const char *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
int _stricoll(const char *s1, const char *s2);
int _wcsicoll(const wchar_t *s1, wconst_t char *s2);
int _mbscoll(const unsigned char *s1, const unsigned char *s2);
int _mbsicoll(const unsigned char *s1, const unsigned char *s2);
```

Description

Compares two strings.

strcoll compares the string pointed to by s1 to the string pointed to by s2, according to the current locale's LC_COLLATE category.

_stricoll performs like strcoll but is not case sensitive.

Note: Note

_stricoll does not compare string according to the current locale's LC_COLLATE category. _stricoll gives you a stricmp. The required collation is obtained by calling _lstricoll, or just plain stricoll (which maps to _lstricoll).

The real collation (_lstricoll) returns -1, 0 or 1, whereas _stricoll does a codepoint comparison, and returns < 0, 0 or > 0.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *two = "International";
   char *one = "Borland";
   int check;
   check = strcoll(one, two);
   if (check == 0)
      printf("The strings are equal\n");
   if (check <  0)
      printf("%s comes before %s\n", one, two);
   if (check >  0)
      printf("%s comes before %s\n", two, one);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcoll	+	+	+	+
_stricoll		+		
_mbscoll		+		
_mbsicoll		+		
wcscoll		+	+	+
_wcsicoll		+		

3.1.4.30.25 strcpy

Header File

string.h, wchar.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strcpy(char *dest, const char *src);
```

```
wchar_t *wcscpy(wchar_t *dest, const wchar_t *src);
```

```
unsigned char *_mbscpy(unsigned char *dest, const unsigned char *src);
```

Description

Copies one string into another.

Copies string src to dest, stopping after the terminating null character has been moved.

Return Value

strcpy returns dest.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
```

```
    char string[10];
    char *str1 = "abcdefghi";
    strcpy(string, str1);
    printf("%s\n", string);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcpy	+	+	+	+
_mbscpy		+		
wcscpy		+	+	+

3.1.4.30.26 strcspn, _mbscspn, wcscspn

Header File

string.h, wchar.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
size_t strcspn(const char *s1, const char *s2);
```

```
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

```
size_t _mbscspn(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the initial segment not containing any subset of a given set of characters.

The strcspn functions search s1 until any one of the characters contained in s2 is found. The number of characters which were read in s1 is the return value. The string termination character is not counted. Neither string is altered during the search.

Return Value

strcspn returns the length of the initial segment of string s1 that consists entirely of characters not from string s2.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
{
    char *string1 = "1234567890";
    char *string2 = "747DC8";
    int length;
    length = strcspn(string1, string2);
    printf("Character where strings intersect is at position %d\n",
           length);
    return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strcspn	+	+	+	+
_mbscspn		+		
wcscspn		+	+	+

3.1.4.30.27 strdup, _mbsdup, _wcsdup

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
char *strdup(const char *s);
```

```
wchar_t *_wcsdup(const wchar_t *s);
```

```
unsigned char *_mbsdup(const wchar_t *s);
```

Description

Copies a string into a newly created location.

strdup makes a duplicate of string s, obtaining space with a call to malloc. The allocated space is (strlen(s) + 1) bytes long. The user is responsible for freeing the space allocated by strdup when it is no longer needed.

Return Value

strdup returns a pointer to the storage location containing the duplicated string, or returns null if space could not be allocated.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
{
   char *dup_str, *string = "abcde";
   dup_str = strdup(string);
   printf("%s\n", dup_str);
   free(dup_str);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strdup		+		
_mbsdup		+		
_wcsdup		+		

3.1.4.30.28 strerror

Header File

string.h

Category

Memory and String Manipulation Routines

Prototype

```
char *strerror(int errnum);
```

Description

Returns a pointer to an error message string.

strerror takes an **int** parameter errnum, an error number, and returns a pointer to an error message string associated with errnum.

Return Value

strerror returns a pointer to a constructed error string. The error message string is constructed in a static buffer that is overwritten with each call to strerror.

Example

```
#include <stdio.h>
#include <errno.h>
int main(void)
{
  char *buffer;
  buffer = strerror(errno);
  printf("Error: %s\n", buffer);
  return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.30.29 stricmp, _mbsicmp, _wcsicmp

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int stricmp(const char *s1, const char *s2);
```

```
int _wcsicmp(const wchar_t *s1, const wchar_t *s2);
```

```
int _mbsicmp(const unsigned char *s1, const unsigned char *s2);
```

Description

Compares one string to another, without case sensitivity.

stricmp performs an unsigned comparison of s1 to s2, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing s1 (or part of it) to s2 (or part of it).

The routines stricmp and strcmpi are the same; strcmpi is implemented through a macro in string.h that translates calls from strcmpi to stricmp. Therefore, in order to use stricmp, you must include the header file string.h for the macro to be available.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *buf1 = "BBB", *buf2 = "bbb";
   int ptr;
   ptr = stricmp(buf2, buf1);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");
   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
stricmp		+	+	+
_mbsicmp		+		
_wcsicmp		+		

3.1.4.30.30 strlwr, _mbslwr, _wcslwr

Header File

string.h, mbstring.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
char *strlwr(char *s);
```

```
wchar_t *_wcslwr(wchar_t *s);
```

```
unsigned char *_mbslwr(unsigned char *s);
```

Description

Converts uppercase letters in a string to lowercase.

strlwr converts uppercase letters in string s to lowercase according to the current locale's LC_CTYPE category. For the C locale, the conversion is from uppercase letters (A to Z) to lowercase letters (a to z). No other characters are changed.

Return Value

strlwr returns a pointer to the string s.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string = "CodeGear";
   printf("string prior to strlwr: %s\n", string);
   strlwr(string);
   printf("string after strlwr:    %s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strlwr		+		
_mbslwr		+		
_wcslwr		+		

3.1.4.30.31 strncat

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strncat(char *dest, const char *src, size_t maxlen);
wchar_t *wcsncat(wchar_t *dest, const wchar_t *src, size_t maxlen);
unsigned char *_mbsncat(unsigned char *dest, const unsigned char *src, size_t maxlen);
unsigned char *_mbsnbcat(unsigned char *__dest, const unsigned char *__src, _SIZE_T __maxlen);
```

Description

Appends a portion of one string to another.

strncat copies at most maxlen characters of src to the end of dest and then appends a null character. The maximum length of the resulting string is strlen(dest) + maxlen.

For _mbsnbcat, if the second byte of 2-bytes character is null, the first byte of this character is regarded as null.

These four functions behave identically and differ only with respect to the type of arguments and return types.

Return Value

strncat returns dest.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char destination[25];
   char *source = " States";
   strcpy(destination, "United");
   strncat(destination, source, 7);
   printf("%s\n", destination);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strncat	+	+	+	+
_mbsncat		+		
_mbsnbcat		+		
_wcsncat		+		

3.1.4.30.32 strncmp, _mbsncmp, wcsncmp

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
int strncmp(const char *s1, const char *s2, size_t maxlen);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t maxlen);
int _mbsncmp(const unsigned char *s1, const unsigned char *s2, size_t maxlen);
#define _mbccmp(__s1, __s2) _mbsncmp((__s1),(__s2),1)
```

Description

Compares a portion of one string to a portion of another.

strncmp makes the same unsigned comparison as strcmp, but looks at no more than maxlen characters. It starts with the first character in each string and continues with subsequent characters until the corresponding characters differ or until it has examined maxlen characters.

Return Value

- strncmp returns an int value based on the result of comparing s1 (or part of it) to s2 (or part of it):
- < 0 if s1 is less than s2
- == 0 if s1 is the same as s2
- > 0 if s1 is greater than s2

Example

```
#include <string.h>
#include <stdio.h>
int  main(void)
{
   char *buf1 = "aaabbb", *buf2 = "bbbccc", *buf3 = "ccc";
   int ptr;
   ptr = strncmp(buf2,buf1,3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");
   ptr = strncmp(buf2,buf3,3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");
   return(0);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strncmp	+	+	+	+
_mbsncmp		+		
_mbccmp		+		
wcsncmp		+	+	+

3.1.4.30.33 strncmpi, wcsncmpi

Header File

string.h

Category

Memory and String Manipulation Routines

Prototype

```
int strncmpi(const char *s1, const char *s2, size_t n);

int wcsncmpi(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

Compares a portion of one string to a portion of another, without case sensitivity.

strncmpi performs a signed comparison of s1 to s2, for a maximum length of n bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until n characters have been examined. The comparison is not case sensitive. (strncmpi is the same as strnicmp--implemented as a macro). It returns a value (< 0, 0, or > 0) based on the result of comparing s1 (or part of it) to s2 (or part of it).

The routines strnicmp and strncmpi are the same; strncmpi is implemented through a macro in string.h that translates calls from strncmpi to strnicmp. Therefore, in order to use strncmpi, you must include the header file string.h for the macro to be available. This macro is provided for compatibility with other C compilers.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *buf1 = "BBBccc", *buf2 = "bbbccc";
   int ptr;
   ptr = strncmpi(buf2,buf1,3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");
   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strncmpi		+		
wcsncmpi		+		

3.1.4.30.34 strncoll, strnicoll, _mbsncoll, _mbsnicoll, _wcsncoll, _wcsnicoll

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int _strncoll(const char *s1, const char *s2, size_t n);
int _wcsncoll(const wchar_t *s1, const wchar_t *s2, size_t n);
int _strnicoll(const char *s1, const char *s2, size_t n);
int _wcsnicoll(const wchar_t *s1, const wchar_t *s2, size_t n);
int _mbsncoll(const unsigned char *s1, const unsigned char *s2, size_t n);
int _mbsnicoll(const unsigned char *s1, const unsigned char *s2, size_t n);
```

Description

_strncoll compares n number of elements from the string pointed to by s1 to the string pointed to by s2, according to the current locale's LC_COLLATE category.

_strnicoll performs like _strncoll but is not case sensitive.

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

3.1.4.30.35 strncpy, _mbsncpy, wcsncpy

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strncpy(char *dest, const char *src, size_t maxlen);
wchar_t *wcsncpy(wchar_t *dest, const wchar_t *src, size_t maxlen);
unsigned char *_mbsncpy(unsigned char *dest, const unsigned char *src, size_t maxlen);
```

Description

Copies a given number of bytes from one string into another, truncating or padding as necessary.

strncpy copies up to maxlen characters from src into dest, truncating or null-padding dest. The target string, dest, might not be null-terminated if the length of src is maxlen or more.

Return Value

strncpy returns dest.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char string[10];
   char *str1 = "abcdefghi";
   strncpy(string, str1, 3);
   string[3] = '\0';
   printf("%s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strncpy	+	+	+	+
_mbsncpy		+		
wcsncpy		+	+	+

3.1.4.30.36 strnicmp, _mbsnicmp, _wcsnicmp

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
int strnicmp(const char *s1, const char *s2, size_t maxlen);
int _wcsnicmp(const wchar_t *s1, const wchar_t *s2, size_t maxlen);
int _mbsnicmp(const unsigned char *s1, const unsigned char *s2, size_t maxlen);
```

Description

Compares a portion of one string to a portion of another, without case sensitivity.

strnicmp performs a signed comparison of s1 to s2, for a maximum length of maxlen bytes, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached. The comparison is not case sensitive.

It returns a value (< 0, 0, or > 0) based on the result of comparing s1 (or part of it) to s2 (or part of it).

Return Value

less than s2	< 0
the same as s2	== 0
greater than s2	> 0

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char *buf1 = "BBBccc", *buf2 = "bbbccc";
   int ptr;
   ptr = strnicmp(buf2, buf1, 3);
   if (ptr > 0)
      printf("buffer 2 is greater than buffer 1\n");
   if (ptr < 0)
      printf("buffer 2 is less than buffer 1\n");
   if (ptr == 0)
      printf("buffer 2 equals buffer 1\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strnicmp		+		
_mbsnicmp		+		
_wcsnicmp		+		

3.1.4.30.37 strnset, _mbsnset, _wcsnset

Header File

string.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strnset(char *s, int ch, size_t n);
wchar_t *_wcsnset(wchar_t *s, wchar_t ch, size_t n);
unsigned char *_mbsnset(unsigned char *s, unsigned int ch, size_t n);
```

Description

Sets a specified number of characters in a string to a given character.

strnset copies the character ch into the first n bytes of the string s. If n > strlen(s), then strlen(s) replaces n. It stops when n characters have been set, or when a null character is found.

Return Value

Each of these functions return s.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string = "abcdefghijklmnopqrstuvwxyz";
   char letter = 'x';
   printf("string before strnset: %s\n", string);
   strnset(string, letter, 13);
   printf("string after  strnset: %s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strnset		+		
_mbsnset		+		
_wcsnset		+		

3.1.4.30.38 strpbrk, _mbspbrk, wcspbrk

Header File

string.h, mbstring.h

Category

C++ Prototyped Routines, Memory and String Manipulation Routines

Prototype

```
char *strpbrk(const char *s1, const char *s2); /* C only */
const char *strpbrk(const char *s1, const char *s2); // C++ only
char *strpbrk(char *s1, const char *s2); // C++ only
wchar_t * wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

```
unsigned char *_mbspbrk(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the first occurrence of any character from a given set.

strpbrk scans a string, s1, for the first occurrence of any character appearing in s2.

Return Value

strpbrk returns a pointer to the first occurrence of any of the characters in s2. If none of the s2 characters occur in s1, strpbrk returns null.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string1 = "abcdefghijklmnopqrstuvwxyz";
   char *string2 = "onm";
   char *ptr;
   ptr = strpbrk(string1, string2);
   if (ptr)
      printf("strpbrk found first character: %c\n", *ptr);
   else
      printf("strpbrk didn't find character in set\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strpbrk	+	+	+	+
_mbspbrk		+		
wcspbrk		+	+	+

3.1.4.30.39 strrchr, _mbsrchr, wcsrchr

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines, C++ Prototyped Routines

Prototype

```
char *strrchr(const char *s, int c); /* C only */
const char *strrchr(const char *s, int c); // C++ only
char *strrchr(char *s, int c); // C++ only
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
unsigned char * _mbsrchr(const unsigned char *s, unsigned int c);
```

Description

Scans a string for the last occurrence of a given character.

strrchr scans a string in the reverse direction, looking for a specific character. strrchr finds the last occurrence of the character c

in the string s. The null-terminator is considered to be part of the string.

Return Value

strrchr returns a pointer to the last occurrence of the character c. If c does not occur in s, strrchr returns null.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char string[15];
   char *ptr, c = 'r';
   strcpy(string, "This is a string");
   ptr = strrchr(string, c);
   if (ptr)
      printf("The character %c is at position: %d\n", c, ptr-string);
   else
      printf("The character was not found\n");
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strrchr	+	+	+	+
_mbsrchr		+		
wcsrchr		+	+	+

3.1.4.30.40 strrev, _mbsrev, _wcsrev

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
char *strrev(char *s);
```

```
wchar_t *_wcsrev(wchar_t *s);
```

```
unsigned char *_mbsrev(unsigned char *s);
```

Description

Reverses a string.

strrev changes all characters in a string to reverse order, except the terminating null character. (For example, it would change string\0 to gnirts\0.)

Return Value

strrev returns a pointer to the reversed string.

Example

```
#include <string.h>
#include <stdio.h>
```

```
int main(void)
{
   char *forward = "string";
   printf("Before strrev(): %s\n", forward);
   strrev(forward);
   printf("After strrev():  %s\n", forward);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strrev		+		
_mbsrev		+		
_wcsrev		+		

3.1.4.30.41 strset, _mbsset, _wcsset

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines, Inline Routines

Prototype

```
char *strset(char *s, int ch);
```

```
wchar_t *_wcsset(wchar_t *s, wchar_t ch);
```

```
unsigned char *_mbsset(unsigned char *s, unsigned int ch);
```

Description

Sets all characters in a string to a given character.

strset sets all characters in the string s to the character ch. It quits when the terminating null character is found.

Return Value

strset returns s.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char string[10] = "123456789";
   char symbol = 'c';
   printf("Before strset(): %s\n", string);
   strset(string, symbol);
   printf("After strset():  %s\n", string);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strset		+		

_mbsset		+		
_wcsset		+		

3.1.4.30.42 strspn, _mbsspn, wcsspn

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
size_t strspn(const char *s1, const char *s2);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
size_t _mbsspn(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the first segment that is a subset of a given set of characters.

strspn finds the initial segment of string s1 that consists entirely of characters from string s2.

Return Value

strspn returns the length of the initial segment of s1 that consists entirely of characters from s2.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
{
   char *string1 = "1234567890";
   char *string2 = "123DC8";
   int length;
   length = strspn(string1, string2);
   printf("Character where strings differ is at position %d\n", length);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strspn	+	+	+	+
_mbsspn		+		
wcsspn		+	+	+

3.1.4.30.43 strstr, _mbsstr, wcsstr

Header File

string.h

Category

C++ Prototyped Routines, Memory and String Manipulation Routines

Prototype

```
char *strstr(const char *s1, const char *s2); /* C only */
const char *strstr(const char *s1, const char *s2); // C++ only
char *strstr(char *s1, const char *s2); // C++ only
wchar_t * wcsstr(const wchar_t *s1, const wchar_t *s2);
unsigned char * _mbsstr(const unsigned char *s1, const unsigned char *s2);
```

Description

Scans a string for the occurrence of a given substring.

strstr scans s1 for the first occurrence of the substring s2.

Return Value

strstr returns a pointer to the element in s1, where s2 begins (points to s2 in s1). If s2 does not occur in s1, strstr returns null.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *str1 = "CodeGear", *str2 = "nation", *ptr;
   ptr = strstr(str1, str2);
   printf("The substring is: %s\n", ptr);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strstr	+	+	+	+
_mbsstr		+		
wcsstr		+	+	+

3.1.4.30.44 strtok, _mbstok, wcstok

Header File

string.h, mbstring.h

Category

Memory and String Manipulation Routines

Prototype

```
char *strtok(char *s1, const char *s2);
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2);
unsigned char *_mbstok(unsigned char *s1, const unsigned char *s2);
```

Description

Searches one string for tokens, which are separated by delimiters defined in a second string.

strtok considers the string s1 to consist of a sequence of zero or more text tokens, separated by spans of one or more characters from the separator string s2.

The first call to strtok returns a pointer to the first character of the first token in s1 and writes a null character into s1 immediately following the returned token. Subsequent calls with null for the first argument will work through the string s1 in this way, until no tokens remain.

The separator string, s2, can be different from call to call.

Note: Calls to strtok cannot be nested with a function call that also uses strtok. Doing so will causes an endless loop.

Return Value

strtok returns a pointer to the token found in s1. A NULL pointer is returned when there are no more tokens.

Example

```
#include <string.h>
#include <stdio.h>
int main(void)
{
   char input[16] = "abc,d";
   char *p;
   /* strtok places a NULL terminator
   in front of the token, if found */
   p = strtok(input, ",");
   if (p)   printf("%s\n", p);
   /* A second call to strtok using a NULL
   as the first parameter returns a pointer
   to the character following the token  */
   p = strtok(NULL, ",");
   if (p)   printf("%s\n", p);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strtok	+	+	+	+
_mbstok		+		
wcstok		+	+	+

3.1.4.30.45 strupr, _mbsupr, _wcsupr

Header File

string.h, mbstring.h

Category

Conversion Routines, Memory and String Manipulation Routines

Prototype

```
char *strupr(char *s);

wchar_t *_wcsupr(wchar_t *s);

unsigned char *_mbsupr(unsigned char *s);
```

Description

Converts lowercase letters in a string to uppercase.

strupr converts lowercase letters in string s to uppercase according to the current locale's LC_CTYPE category. For the default C locale, the conversion is from lowercase letters (a to z) to uppercase letters (A to Z). No other characters are changed.

Return Value

strupr returns s.

Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char *string = "abcdefghijklmnopqrstuvwxyz", *ptr;
   /* converts string to upper case characters */
   ptr = strupr(string);
   printf("%s\n", ptr);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strupr		+		
_mbsupr		+		
_wcsupr		+		

3.1.4.30.46 strxfrm, wcsxfrm

Header File

string.h

Category

Memory and String Manipulation Routines

Prototype

```
size_t strxfrm(char *target, const char *source, size_t n);
```

```
size_t wcsxfrm(wchar_t *target, const wchar_t *source, size_t n);
```

Description

Transforms a portion of a string to a specified collation.

strxfrm transforms the string pointed to by source into the string target for no more than n characters. The transformation is such that if the strcmp function is applied to the resulting strings, its return corresponds with the return values of the strcoll function.

No more than n characters, including the terminating null character, are copied to target.

strxfrm transforms a character string into a special string according to the current locale's LC_COLLATE category. The special string that is built can be compared with another of the same type, byte for byte, to achieve a locale-correct collation result. These special strings, which can be thought of as keys or tokenized strings, are not compatible across the different locales.

The tokens in the tokenized strings are built from the collation weights used by strcoll from the active locale's collation tables.

Processing stops only after all levels have been processed for the character string or the length of the tokenized string is equal to the maxlen parameter.

All redundant tokens are removed from each level's set of tokens.

The tokenized string buffer must be large enough to contain the resulting tokenized string. The length of this buffer depends on the size of the character string, the number of collation levels, the rules for each level and whether there are any special characters in the character string. Certain special characters can cause extra character processing of the string resulting in more space requirements. For example, the French character "oe" will take double the space for itself because in some locales, it expands to collation weights for each level. Substrings that have substitutions will also cause extra space requirements.

There is no safe formula to determine the required string buffer size, but at least (levels * string length) are required.

Return Value

Number of characters copied not including the terminating null character. If the value returned is greater than or equal to n, the content of target is indeterminate.

Example

```
#include <stdio.h>
#include <string.h>
#include <alloc.h>
int main(void)
{
   char *target;
   char *source = "Frank Borland";
   int length;
   /* allocate space for the target string */
   target = (char *) calloc(80, sizeof(char));
   /* copy the source over to the target and get the length */
   length = strxfrm(target, source, 80);
   /* print out the results */
   printf("%s has the length %d\n", target, length);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strxfrm	+	+	+	+
wcsxfrm		+	+	+

3.1.4.31 sys\stat.h

The following functions, macros, and classes are provided in `sys\stat.h`:

Topics

Name	Description
S_Ixxxx #defines (see page 1187)	**Header File** sys\stat.h **Description** Definitions used for file status and directory functions.

_stati64, _tstati64, stati64, _wstati64 (see page 1188)	**Header File**
	sys\stat.h, tchar.h
	Category
	Directory Control Routines
	Prototype
	`int stati64(const char *pathname, struct stati64 *buff);`
	`int _stati64(const char *__path, struct stati64 *__statbuf);`
	`int _wstati64(const wchar_t *pathname, struct stati64 *buff);`
	`// From tchar.h`
	`#define _tstati64 _stati64`
	Description
	Gather statistics about the file named by *pathP and place them in the buffer *bufP.
	The statistics fields are set thus:
	st_devset to -1 if S_IFCHR, else set to drive holding the file.
	st_ino0
	st_modeUnix-style bit-set for file access rights
	st_nlink1
	st_uid0
	st_gid0
	st_rdevsame as st_dev
	st_sizefile size (0 if S_IFDIR or S_IFCHR)
	st_atimetime file last changed (seconds since 1970)
	st_mtimesame as st_atime... more (see page 1188)

fstat, stat, _wstat (see page 1189)	**Header File**
	sys\stat.h
	Category
	Input/output Routines
	Prototype
	`int fstat(int handle, struct stat *statbuf);`
	`int stat(const char *path, struct stat *statbuf);`
	`int _wstat(const wchar_t *path, struct stat *statbuf);`
	Description
	Gets open file information.
	fstat stores information in the stat structure about the file or directory associated with handle.
	stat stores information about a given file or directory in the stat structure. The name of the file is path.
	statbuf points to the stat structure (defined in sys\stat.h). That structure contains the following fields:

3.1.4.31.1 S_Ixxxx #defines

Header File

sys\stat.h

Description

Definitions used for file status and directory functions.

Name	Meaning
S_IFMT	File type mask
S_IFDIR	Directory
S_IFIFO	FIFO special
S_IFCHR	Character special
S_IFBLK	Block special
S_IFREG	Regular file

S_IREAD	Owner can read
S_IWRITE	Owner can write
S_IEXEC	Owner can execute

3.1.4.31.2 _stati64, _tstati64, stati64, _wstati64

Header File

sys\stat.h, tchar.h

Category

Directory Control Routines

Prototype

```
int stati64(const char *pathname, struct stati64 *buff);
int _stati64(const char *__path, struct stati64 *__statbuf);
int _wstati64(const wchar_t *pathname, struct stati64 *buff);
// From tchar.h
#define _tstati64 _stati64
```

Description

Gather statistics about the file named by *pathP and place them in the buffer *bufP.

The statistics fields are set thus:

st_devset to -1 if S_IFCHR, else set to drive holding the file.

st_ino0

st_modeUnix-style bit-set for file access rights

st_nlink1

st_uid0

st_gid0

st_rdevsame as st_dev

st_sizefile size (0 if S_IFDIR or S_IFCHR)

st_atimetime file last changed (seconds since 1970)

st_mtimesame as st_atime

st_ctimesame as st_atime

The file access rights bit-set may contain S_IFCHR, S_IFDIR, S_IFREG, S_IREAD, S_IWRITE, or S_IEXEC.

If the name is for a device, the time fields will be zero and the size field is undefined.

Return Value

The return value is 0 if the call was successful, otherwise -1 is returned and errno contains the reason. The buffer is not touched unless the call is successful.

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.31.3 fstat, stat, _wstat

Header File

sys\stat.h

Category

Input/output Routines

Prototype

```
int fstat(int handle, struct stat *statbuf);
int stat(const char *path, struct stat *statbuf);
int _wstat(const wchar_t *path, struct stat *statbuf);
```

Description

Gets open file information.

fstat stores information in the stat structure about the file or directory associated with handle.

stat stores information about a given file or directory in the stat structure. The name of the file is path.

statbuf points to the stat structure (defined in sys\stat.h). That structure contains the following fields:

st_mode	Bit mask giving information about the file's mode
st_dev	Drive number of disk containing the file or file handle if the file is on a device
st_rdev	Same as st_dev
st_nlink	Set to the integer constant 1
st_size	Size of the file in bytes
st_atime	Most recent access (Windows) or last time modified (DOS)
st_mtime	Same as st_atime
st_ctime	Same as st_atime

The stat structure contains three more fields not mentioned here. They contain values that are meaningful only in UNIX.

The st_mode bit mask that gives information about the mode of the open file includes the following bits:

One of the following bits will be set:

S_IFCHR	If handle refers to a device.
S_IFREG	If an ordinary file is referred to by handle.

One or both of the following bits will be set:

S_IWRITE	If user has permission to write to file.
S_IREAD	If user has permission to read to file.

The HPFS and NTFS file-management systems make the following distinctions:

st_atime	Most recent access
st_mtime	Most recent modify
st_ctime	Creation time

Return Value

fstat and stat return 0 if they successfully retrieved the information about the open file.

On error (failure to get the information) these functions return -1 and set the global variable errno to

EBADF	Bad file handle

Example

```
#include <sys\stat.h>
#include <stdio.h>
#include <time.h>
int main(void)
{
   struct stat statbuf;
   FILE *stream;
   /* open a file for update */
   if ((stream = fopen("DUMMY.FIL", "w+"))
       == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
      return(1);
   }
   fprintf(stream, "This is a test");
   fflush(stream);
   /* get information about the file */
   fstat(fileno(stream), &statbuf);
   fclose(stream);
   /* display the information returned */
   if (statbuf.st_mode & S_IFCHR)
      printf("Handle refers to a device.\n");
   if (statbuf.st_mode & S_IFREG)
      printf("Handle refers to an ordinary file.\n");
   if (statbuf.st_mode & S_IREAD)
      printf("User has read permission on file.\n");
   if (statbuf.st_mode & S_IWRITE)
      printf("User has write permission on file.\n");
   printf("Drive letter of file: %c\n", 'A'+statbuf.st_dev);
   printf("Size of file in bytes: %ld\n", statbuf.st_size);
   printf("Time file last opened: %s\n", ctime(&statbuf.st_ctime));
   return 0;
}
```

3.1.4.32 sys\timeb.h

The following functions, macros, and classes are provided in `sys\timeb.h`:

Topics

Name	Description

ftime (see page 1191)	**Header File**
	sys\timeb.h
	Category
	Time and Date Routines
	Prototype
	`void ftime(struct timeb *buf)`
	Description
	Stores current time in timeb structure.
	On UNIX platforms ftime is available only on System V systems.
	ftime determines the current time and fills in the fields in the timeb structure pointed to by buf. The timeb structure contains four fields: time, millitm, _timezone, and dstflag:
	`struct timeb {`
	`long time ;`
	`short millitm ;`
	`short _timezone ;`
	`short dstflag ;`
	`};`
	• timeprovides the time in seconds since 00:00:00 Greenwich mean time (GMT) January 1 1970.
	• millitmis the fractional part of a second in milliseconds.... more (see page 1191)

3.1.4.32.1 ftime

Header File

sys\timeb.h

Category

Time and Date Routines

Prototype

```
void ftime(struct timeb *buf)
```

Description

Stores current time in timeb structure.

On UNIX platforms ftime is available only on System V systems.

ftime determines the current time and fills in the fields in the timeb structure pointed to by buf. The timeb structure contains four fields: time, millitm, _timezone, and dstflag:

```
struct timeb {
long time ;
short millitm ;
short _timezone ;
short dstflag ;
};
```

- timeprovides the time in seconds since 00:00:00 Greenwich mean time (GMT) January 1 1970.
- millitmis the fractional part of a second in milliseconds.

- _timezoneis the difference in minutes between GMT and the local time. This value is computed going west from GMT. ftime gets this field from the global variable _timezone which is set by tzset.
- dstflagis set to nonzero if daylight saving time is taken into account during time calculations.

  Note: ftime calls tzset. Therefore it isn't necessary to call tzset explicitly when you use ftime.

  Return Value

  None.

  Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys\timeb.h>
/* pacific standard & daylight savings */
char *tzstr = "TZ=PST8PDT";
int main(void)
{
   struct timeb t;
   putenv(tzstr);
   tzset();
   ftime(&t);
   printf("Seconds since 1/1/1970 GMT: %ld\n", t.time);
   printf("Thousandths of a second: %d\n", t.millitm);
   printf("Difference between local time and GMT: %d\n", t._timezone);
   printf("Daylight savings in effect (1) not (0): %d\n", t.dstflag);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.33 sys\types.h

The following functions, macros, and classes are provided in `sys\types.h`:

Topics

Name	Description
time_t (see page 1192)	**Header File** sys\types.h time.h **Syntax** `typedef long time_t;` **Description** Defines the value used by the time functions declared in time.h.

3.1.4.33.1 time_t

Header File

sys\types.h

time.h

Syntax

```
typedef long time_t;
```

Description

Defines the value used by the time functions declared in time.h.

3.1.4.34 time.h

The following functions, macros, and classes are provided in `time.h`:

Topics

Name	Description

_strdate, _wstrdate (see page 1197)	**Header File**
	time.h
	Category
	Conversion Routines, Time and Date Routines
	Prototype
	`char *_strdate(char *buf);`
	`wchar_t *_wstrdate(wchar_t *buf);`
	Description
	Converts current date to string.
	_strdate converts the current date to a string, storing the string in the buffer buf. The buffer must be at least 9 characters long.
	The string has the form MM/DD/YY where MM, DD, and YY are all two-digit numbers representing the month, day, and year. The string is terminated by a null character.
	Return Value
	_strdate returns buf, the address of the date string.
	Example

_strtime, _wstrtime (see page 1198)	**Header File**
	time.h
	Category
	Time and Date Routines
	Prototype
	`char *_strtime(char *buf);`
	`wchar_t *_wstrtime(wchar_t *buf);`
	Description
	Converts current time to string.
	_strtime converts the current time to a string, storing the string in the buffer buf. The buffer must be at least 9 characters long.
	The string has the following form:
	`HH:MM:SS`
	where HH, MM, and SS are all two-digit numbers representing the hour, minute, and second, respectively. The string is terminated by a null character.
	Return Value
	_strtime returns buf, the address of the time string.
	Example

asctime (see page 1199)	**Header File**
	time.h
	Category
	Time and Date Routines
	Prototype
	`char *asctime(const struct tm *tblock);`
	`wchar_t *_wasctime(const struct tm *tblock);`
	Description
	asctime converts date and time to ASCII.
	_wasctime converts date and time to a **wchar_t** string.
	asctime and _wasctime convert a time stored as a structure to a 26 (wide) character string in the following form:
	Mon Nov 21 11:31:54 1983\n\0
	All the fields have a constant width. The output string day-of-the-week and month correspond to the following:
	tm parameterValid value rangeOutput
	tm.mon (month)0-110=Jan, 1=Feb, and so on
	tm.day (day-of-the-week)0-60=Sun, 1=Mon, and so on
	Return Value
	asctime and _wasctime return... more (see page 1199)

clock (see page 1200)	**Header File** time.h **Category** Time and Date Routines **Prototype** `clock_t clock(void);` **Description** Determines processor time. clock can be used to determine the time interval between two events. To determine the time in seconds, the value returned by clock should be divided by the value of the macro CLK_TCK. **Return Value** On success, clock returns the processor time elapsed since the beginning of the program invocation. On error (if the processor time is not available or its value cannot be represented), clock returns -1. **Example**
clock_t (see page 1201)	**Header File** time.h **Syntax** `typedef long clock_t;` **Description** Defines the data type returned by the clock function. **Portability**

ctime, _wctime (see page 1201)	**Header File**
	time.h
	Category
	Time and Date Routines
	Prototype
	`char *ctime(const time_t *time);`
	`wchar_t *_wctime(const time_t *time);`
	Description
	Converts date and time to a string.
	ctime converts a time value pointed to by time (the value returned by the function time) into a 26-character string in the following form, terminating with a newline character and a null character:
	`Mon Nov 21 11:31:54 1983\n\0`
	All the fields have constant width.
	The global long variable _timezone contains the difference in seconds between GMT and local standard time (in PST, _timezone is 8*60*60). The global variable _daylight is used to tell the RTL's... more (see page 1201)

_daylight (see page 1202)	**Header File** time.h **Syntax** `extern int _daylight;` **Description** _daylight is used by the time and date functions. It is used to tell the RTL's functions (mktime & localtime) whether they should take daylight saving time into account if it runs into a date that would normally fall into that category. _daylight is initialized from the values specified in the TZ environment variable and is set to 1 if the daylight savings time conversion should be applied. If TZ is not set, the value of _daylight is obtained from the operating system.
difftime (see page 1202)	**Header File** time.h **Category** Time and Date Routines **Prototype** `double difftime(time_t time2, time_t time1);` **Description** Computes the difference between two times. difftime calculates the elapsed time in seconds, from time1 to time2. **Return Value** difftime returns the result of its calculation as a **double**. **Example**
gmtime (see page 1203)	**Header File** time.h **Category** Time and Date Routines **Prototype** `struct tm *gmtime(const time_t *timer);` **Description** Converts date and time to Greenwich mean time (GMT). gmtime accepts the address of a value returned by time and returns a pointer to the structure of type tm containing the time elements. gmtime converts directly to GMT. The global long variable _timezone should be set to the difference in seconds between GMT and local standard time (in PST _timezone is 8 x 60 x 60). The global variable _daylight should be set to nonzero only if the standard U.S. daylight saving time conversion should... more (see page 1203)
localtime (see page 1204)	**Header File** time.h **Category** Time and Date Routines **Prototype** `struct tm *localtime(const time_t *timer);` **Description** Converts date and time to a structure. localtime accepts the address of a value returned by time and returns a pointer to the structure of type tm containing the time elements. It corrects for the time zone and possible daylight saving time. The global long variable _timezone contains the difference in seconds between GMT and local standard time (in PST, _timezone is 8 x 60 x 60). The global variable _daylight is used to tell the RTL's functions (mktime & localtime) whether they should take... more (see page 1204)

mktime (see page 1206)	**Header File** time.h **Category** Time and Date Routines **Prototype** `time_t mktime(struct tm *t);` **Description** Converts time to calendar format. Converts the time in the structure pointed to by t into a calendar time with the same format used by the time function. The original values of the fields tm_sec, tm_min, tm_hour, tm_mday, and tm_mon are not restricted to the ranges described in the tm structure. If the fields are not in their proper ranges, they are adjusted. Values for fields tm_wday and tm_yday are computed after the other fields have been adjusted. The tm_isdst (Daylight Savings Time) field is adjusted... more (see page 1206)
stime (see page 1207)	**Header File** time.h **Category** Time and Date Routines **Prototype** `int stime(time_t *tp);` **Description** Sets system date and time. stime sets the system time and date. tp points to the value of the time as measured in seconds from 00:00:00 GMT, January 1, 1970. **Return Value** stime returns a value of 0. **Example**

strftime, wcsftime (see page 1207)	**Header File**
	time.h
	Category
	Time and Date Routines
	Prototype
	`size_t strftime(char *s, size_t maxsize, const char *fmt, const struct tm *t);`
	`size_t wcsftime(wchar_t *s, size_t maxsize, const wchar_t *fmt, const struct tm *t);`
	Description
	Formats time for output.
	strftime formats the time in the argument t into the array pointed to by the argument s according to the fmt specifications. All ordinary characters are copied unchanged. No more than maxsize characters are placed in s.
	The time is formatted according to the current locale's LC_TIME category.
	Return Value
	On success, strftime returns the number of characters placed into s.
	On... more (see page 1207)

time (see page 1208)	**Header File** time.h **Category** Time and Date Routines **Prototype** `time_t time(time_t *timer);` **Description** Gets time of day. time gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970, and stores that value in the location pointed to by timer, provided that timer is not a NULL pointer. **Return Value** time returns the elapsed time in seconds. **Example**

_timezone (see page 1209)	**Header File** time.h **Syntax** `extern long _timezone;` **Description** _timezone is used by the time-and-date functions. It is calculated by the tzset function; it is assigned a long value that is the difference, in seconds, between the current local time and Greenwich mean time. On Win32, the value of _timezone is obtained from the operating system.

tm (see page 1209)	**Header File**
	time.h
	Syntax
	`struct tm {`
	`int tm_sec; /* Seconds */`
	`int tm_min; /* Minutes */`
	`int tm_hour; /* Hour (0--23) */`
	`int tm_mday; /* Day of month (1--31) */`
	`int tm_mon; /* Month (0--11) */`
	`int tm_year; /* Year (calendar year minus 1900) */`
	`int tm_wday; /* Weekday (0--6; Sunday = 0) */`
	`int tm_yday; /* Day of year (0--365) */`
	`int tm_isdst; /* 0 if daylight savings time is not in effect) */`
	`};`
	Description
	A structure defining the time, broken down into increments.
	tm is used by the functions asctime, gmtime, localtime, mktime, and strftime.
	Example... more (see page 1209)

_tzname,_wtzname (see page 1210)	**Header File**
	time.h
	Syntax
	`extern char * _tzname[2]`
	`extern wchar_t *const _wtzname[2]`
	Description
	The global variable _tzname is an array of pointers to strings containing abbreviations for time zone names. _tzname[0] points to a three-character string with the value of the time zone name from the TZ environment string. The global variable _tzname[1] points to a three-character string with the value of the daylight saving time zone name from the TZ environment string. If no daylight saving name is present, _tzname[1] points to a null string.
	On Win32, the value of _tzname is obtained from the operating system.

_tzset, _wtzset (see page 1210)	**Header File**
	time.h
	Category
	Time and Date Routines
	Prototype
	`void _tzset(void)`
	`void _wtzset(void)`
	Description
	Sets value of global variables _daylight, _timezone, and _tzname.
	_tzset is available on XENIX systems.
	_tzset sets the _daylight, _timezone, and _tzname global variables based on the environment variable TZ. _wtzset sets the _daylight, _timezone, and _wtzname global variables. The library functions ftime and localtime use these global variables to adjust Greenwich Mean Time (GMT) to the local time zone. The format of the TZ environment string is:
	`TZ = zzz[+/-]d[d][lll]`
	where zzz is a three-character string representing the name of the current time zone. All... more (see page 1210)

3.1.4.34.1 _strdate, _wstrdate

Header File

time.h

Category

Conversion Routines, Time and Date Routines

Prototype

```
char *_strdate(char *buf);
```

```
wchar_t *_wstrdate(wchar_t *buf);
```

Description

Converts current date to string.

_strdate converts the current date to a string, storing the string in the buffer buf. The buffer must be at least 9 characters long.

The string has the form MM/DD/YY where MM, DD, and YY are all two-digit numbers representing the month, day, and year. The string is terminated by a null character.

Return Value

_strdate returns buf, the address of the date string.

Example

```
#include <time.h>
#include <stdio.h>
void main(void)
{
  char datebuf[9];
  char timebuf[9];
  _strdate(datebuf);
  _strtime(timebuf);
  printf("Date: %s  Time: %s\n",datebuf,timebuf);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strdate		+		
_wstrdate		+		

3.1.4.34.2 _strtime, _wstrtime

Header File

time.h

Category

Time and Date Routines

Prototype

```
char *_strtime(char *buf);
```

```
wchar_t *_wstrtime(wchar_t *buf);
```

Description

Converts current time to string.

_strtime converts the current time to a string, storing the string in the buffer buf. The buffer must be at least 9 characters long.

The string has the following form:

`HH:MM:SS`

where HH, MM, and SS are all two-digit numbers representing the hour, minute, and second, respectively. The string is terminated by a null character.

Return Value

_strtime returns buf, the address of the time string.

Example

```
#include <time.h>
#include <stdio.h>
void main(void)
{
  char datebuf[9];
  char timebuf[9];
  _strdate(datebuf);
  _strtime(timebuf);
  printf("Date: %s  Time: %s\n",datebuf,timebuf);
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_strtime		+		
_wstrtime		+		

3.1.4.34.3 asctime

Header File

time.h

Category

Time and Date Routines

Prototype

```
char *asctime(const struct tm *tblock);
```

```
wchar_t *_wasctime(const struct tm *tblock);
```

Description

asctime converts date and time to ASCII.

_wasctime converts date and time to a **wchar_t** string.

asctime and _wasctime convert a time stored as a structure to a 26 (wide) character string in the following form:

Mon Nov 21 11:31:54 1983\n\0

All the fields have a constant width. The output string day-of-the-week and month correspond to the following:

tm parameterValid value rangeOutput

tm.mon (month)0-110=Jan, 1=Feb, and so on

tm.day (day-of-the-week)0-60=Sun, 1=Mon, and so on

Return Value

asctime and _wasctime return a pointer to the (wide) character string containing the date and time. This string is a static which is overwritten with each call. asctime converts a time stored as a structure in *tblock to a 26-character string of the same form as the ctime string:

```
Sun Sep 16 01:03:52 1973\n\0
```

Example

```
#include <string.h>
#include <time.h>
#include <stdio.h>
int main(void)
{
  struct tm t;
  char str[80];
  /* sample loading of tm structure  */
  t.tm_sec    = 1;  /* Seconds */
  t.tm_min    = 30; /* Minutes */
  t.tm_hour   = 9;  /* Hour */
  t.tm_mday   = 22; /* Day of the Month  */
  t.tm_mon    = 11; /* Month */
  t.tm_year   = 56; /* Year - does not include century */
  t.tm_wday   = 4;  /* Day of the week  */
  t.tm_yday   = 0;  /* Does not show in asctime */
  t.tm_isdst  = 0;  /* Is Daylight SavTime; does not show in asctime */
  /* converts structure to null terminated string */
  strcpy(str, asctime(&t));
  printf("%s\n", str);
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
asctime	+	+	+	+
_wasctime		+		

3.1.4.34.4 clock

Header File

time.h

Category

Time and Date Routines

Prototype

```
clock_t clock(void);
```

Description

Determines processor time.

clock can be used to determine the time interval between two events. To determine the time in seconds, the value returned by clock should be divided by the value of the macro CLK_TCK.

Return Value

On success, clock returns the processor time elapsed since the beginning of the program invocation.

On error (if the processor time is not available or its value cannot be represented), clock returns -1.

Example

```
#include <time.h>
#include <stdio.h>
#include <dos.h>
int main(void)
{
   clock_t start, end;
   start = clock();
   delay(2000);
   end = clock();
   printf("The time was: %f\n", (end - start) / CLK_TCK);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.34.5 clock_t

Header File

time.h

Syntax

```
typedef long clock_t;
```

Description

Defines the data type returned by the clock function.

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.34.6 ctime, _wctime

Header File

time.h

Category

Time and Date Routines

Prototype

```
char *ctime(const time_t *time);
```

```
wchar_t *_wctime(const time_t *time);
```

Description

Converts date and time to a string.

ctime converts a time value pointed to by time (the value returned by the function time) into a 26-character string in the following form, terminating with a newline character and a null character:

```
Mon Nov 21 11:31:54 1983\n\0
```

All the fields have constant width.

The global long variable _timezone contains the difference in seconds between GMT and local standard time (in PST, _timezone is 8*60*60). The global variable _daylight is used to tell the RTL's functions (mktime & localtime) whether they should take daylight saving time into account if it runs into a date that would normally fall into that category. It is set to 1 if the daylight savings time conversion should be applied.. These variables are set by the tzset function, not by the user program directly.

Return Value

ctime returns a pointer to the character string containing the date and time. The return value points to static data that is overwritten with each call to ctime.

Example

```
#include <stdio.h>
#include <time.h>
int main(void)
{
  time_t t;
  time(&t);
  printf("Today's date and time: %s\n", ctime(&t));
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
ctime	+	+	+	+
_wctime		+		

3.1.4.34.7 _daylight

Header File

time.h

Syntax

```
extern int _daylight;
```

Description

_daylight is used by the time and date functions. It is used to tell the RTL's functions (mktime & localtime) whether they should take daylight saving time into account if it runs into a date that would normally fall into that category. _daylight is initialized from the values specified in the TZ environment variable and is set to 1 if the daylight savings time conversion should be applied. If TZ is not set, the value of _daylight is obtained from the operating system.

3.1.4.34.8 difftime

Header File

time.h

Category

Time and Date Routines

Prototype

```
double difftime(time_t time2, time_t time1);
```

Description

Computes the difference between two times.

difftime calculates the elapsed time in seconds, from time1 to time2.

Return Value

difftime returns the result of its calculation as a **double**.

Example

```
#include <time.h>
#include <stdio.h>
#include <dos.h>
#include <conio.h>
int main(void)
{
   time_t first, second;
   clrscr();
   first = time(NULL);  /* Gets system
                           time */
   delay(2000);         /* Waits 2 secs */
   second = time(NULL); /* Gets system time
                           again */
   printf("The difference is: %f seconds\n",difftime(second,first));
   getch();
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.34.9 gmtime

Header File

time.h

Category

Time and Date Routines

Prototype

```
struct tm *gmtime(const time_t *timer);
```

Description

Converts date and time to Greenwich mean time (GMT).

gmtime accepts the address of a value returned by time and returns a pointer to the structure of type tm containing the time elements. gmtime converts directly to GMT.

The global long variable _timezone should be set to the difference in seconds between GMT and local standard time (in PST _timezone is 8 x 60 x 60). The global variable _daylight should be set to nonzero only if the standard U.S. daylight saving time

conversion should be applied.

This is the tm structure declaration from the time.h header file:

```
struct tm {
int tm_sec; /* Seconds */
int tm_min; /* Minutes */
int tm_hour; /* Hour (0 - 23) */
int tm_mday; /* Day of month (1 - 31) */
int tm_mon; /* Month (0 - 11) */
int tm_year; /* Year (calendar year minus 1900) */
int tm_wday; /* Weekday (0 - 6; Sunday is 0) */
int tm_yday; /* Day of year (0 -365) */
int tm_isdst; /* Nonzero if daylight saving time is in effect. */
};
```

These quantities give the time on a 24-hour clock, day of month (1 to 31), month (0 to 11), weekday (Sunday equals 0), year - 1900, day of year (0 to 365), and a flag that is nonzero if daylight saving time is in effect.

Return Value

gmtime returns a pointer to the structure containing the time elements. This structure is a static that is overwritten with each call.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
/* Pacific Standard Time & Daylight Savings */
char *tzstr = "TZ=PST8PDT";
int main(void)
{
   time_t t;
   struct tm *gmt, *area;
   putenv(tzstr);
   tzset();
   t = time(NULL);
   area = localtime(&t);
   printf("Local time is: %s", asctime(area));
   gmt = gmtime(&t);
   printf("GMT is:        %s", asctime(gmt));
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.34.10 localtime

Header File

time.h

Category

Time and Date Routines

Prototype

```
struct tm *localtime(const time_t *timer);
```

Description

Converts date and time to a structure.

localtime accepts the address of a value returned by time and returns a pointer to the structure of type tm containing the time elements. It corrects for the time zone and possible daylight saving time.

The global long variable _timezone contains the difference in seconds between GMT and local standard time (in PST, _timezone is 8 x 60 x 60). The global variable _daylight is used to tell the RTL's functions (mktime & localtime) whether they should take daylight saving time into account if it runs into a date that would normally fall into that category. It is set to 1 if the daylight savings time conversion should be applied. These values are set by tzset, not by the user program directly.

This is the **tm** structure declaration from the time.h header file:

```
struct tm {
int tm_sec;
int tm_min;
int tm_hour;
int tm_mday;
int tm_mon;
int tm_year;
int tm_wday;
int tm_yday;
int tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1 to 31), month (0 to 11), weekday (Sunday equals 0), year - 1900, day of year (0 to 365), and a flag that is nonzero if the daylight saving time conversion should be applied.

Return Value

localtime returns a pointer to the structure containing the time elements. This structure is a static that is overwritten with each call.

Example

```
#include <time.h>
#include <stdio.h>
int main(void)
{
   time_t timer;
   struct tm *tblock;
   /* gets time of day */
   timer = time(NULL);
   /* converts date/time to a structure */
   tblock = localtime(&timer);
   printf("Local time is: %s", asctime(tblock));
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.34.11 mktime

Header File

time.h

Category

Time and Date Routines

Prototype

```
time_t mktime(struct tm *t);
```

Description

Converts time to calendar format.

Converts the time in the structure pointed to by t into a calendar time with the same format used by the time function. The original values of the fields tm_sec, tm_min, tm_hour, tm_mday, and tm_mon are not restricted to the ranges described in the tm structure. If the fields are not in their proper ranges, they are adjusted. Values for fields tm_wday and tm_yday are computed after the other fields have been adjusted.

The tm_isdst (Daylight Savings Time) field is adjusted with the correct value after calling the function mktime. Also, tm_isdst is used for adusting the value of tm_hour. For example, if the value of tm_isdst is 1, but the current date is not DST, mktime updates tm_hour (tm_hour = tm_hour - 1) and sets tm_isdst to 0.

The allowable range of calendar times is Jan 1 1970 00:00:00 to Jan 19 2038 03:14:07.

Return Value

On success, mktime returns calendar time as described above.

On error (if the calendar time cannot be represented), mktime returns -1.

Example

```
#include <stdio.h> #include <time.h>
        char *wday[] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
        "Saturday", "Unknown"}; int main(void) { struct tm time_check; int year, month, day; /*
        Input a year, month and day to find the weekday for */ printf("Year: "); scanf("%d",
        &year); printf("Month: "); scanf("%d", &month); printf("Day: "); scanf("%d",
        &day); /* load the time_check structure with the data */ time_check.tm_year = year -
        1900; time_check.tm_mon = month - 1; time_check.tm_mday = day; time_check.tm_hour = 0;
        time_check.tm_min = 0; time_check.tm_sec = 1; time_check.tm_isdst = -1; /* call mktime
to
        fill in the weekday field of the structure */ if (mktime(&time_check) == -1)
        time_check.tm_wday = 7; /* print out the day of the week */ printf("That day is a
%s\n",
        wday[time_check.tm_wday]); return 0; }
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.34.12 stime

Header File

time.h

Category

Time and Date Routines

Prototype

```
int stime(time_t *tp);
```

Description

Sets system date and time.

stime sets the system time and date. tp points to the value of the time as measured in seconds from 00:00:00 GMT, January 1, 1970.

Return Value

stime returns a value of 0.

Example

```
#include <stdio.h>
#include <time.h>
int main(void)
{
  time_t t;
  t = time(NULL);
  printf("Current date is %s", ctime(&t));
  t -= 24L*60L*60L;  /* Back up to same time previous day */
  stime(&t);
  printf("\nNew date is %s", ctime(&t));
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
	+		

3.1.4.34.13 strftime, wcsftime

Header File

time.h

Category

Time and Date Routines

Prototype

```
size_t strftime(char *s, size_t maxsize, const char *fmt, const struct tm *t);
```

```
size_t wcsftime(wchar_t *s, size_t maxsize, const wchar_t *fmt, const struct tm *t);
```

Description

Formats time for output.

strftime formats the time in the argument t into the array pointed to by the argument s according to the fmt specifications. All ordinary characters are copied unchanged. No more than maxsize characters are placed in s.

The time is formatted according to the current locale's LC_TIME category.

Return Value

On success, strftime returns the number of characters placed into s.

On error (if the number of characters required is greater than maxsize), strftime returns 0.

More about strftime

Example

```
#include <stdio.h>
#include <time.h>
#include <dos.h>
int main(void)
{
   struct tm *time_now;
   time_t secs_now;
   char str[80];
   tzset();
   time(&secs_now);
   time_now = localtime(&secs_now);
   strftime(str, 80,
            "It is %M minutes after %I o'clock (%Z)  %A, %B %d 19%y",
            time_now);
   printf("%s\n",str);
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
strftime	+	+	+	+
wcsftime		+	+	+

3.1.4.34.14 time

Header File

time.h

Category

Time and Date Routines

Prototype

```
time_t time(time_t *timer);
```

Description

Gets time of day.

time gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 1970, and stores that value in the location pointed to by timer, provided that timer is not a NULL pointer.

Return Value

time returns the elapsed time in seconds.

Example

```
#include <time.h>
#include <stdio.h>
int main(void)
{
   time_t t;
   t = time(NULL);
   printf("The number of seconds since January 1, 1970 is %ld",t);
   return 0;
}
```

Portability

POSIX	Win32	ANSI C	ANSI C++
+	+	+	+

3.1.4.34.15 _timezone

Header File

time.h

Syntax

```
extern long _timezone;
```

Description

_timezone is used by the time-and-date functions. It is calculated by the tzset function; it is assigned a long value that is the difference, in seconds, between the current local time and Greenwich mean time.

On Win32, the value of _timezone is obtained from the operating system.

3.1.4.34.16 tm

Header File

time.h

Syntax

```
struct tm {
int tm_sec; /* Seconds */
int tm_min; /* Minutes */
int tm_hour; /* Hour (0--23) */
int tm_mday; /* Day of month (1--31) */
int tm_mon; /* Month (0--11) */
int tm_year; /* Year (calendar year minus 1900) */
int tm_wday; /* Weekday (0--6; Sunday = 0) */
int tm_yday; /* Day of year (0--365) */
int tm_isdst; /* 0 if daylight savings time is not in effect) */
```

```
};
```

Description

A structure defining the time, broken down into increments.

tm is used by the functions asctime, gmtime, localtime, mktime, and strftime.

Example

3.1.4.34.17 _tzname,_wtzname

Header File

time.h

Syntax

```
extern char * _tzname[2]
```

```
extern wchar_t *const _wtzname[2]
```

Description

The global variable _tzname is an array of pointers to strings containing abbreviations for time zone names. _tzname[0] points to a three-character string with the value of the time zone name from the TZ environment string. The global variable _tzname[1] points to a three-character string with the value of the daylight saving time zone name from the TZ environment string. If no daylight saving name is present, _tzname[1] points to a null string.

On Win32, the value of _tzname is obtained from the operating system.

3.1.4.34.18 _tzset, _wtzset

Header File

time.h

Category

Time and Date Routines

Prototype

```
void _tzset(void)
```

```
void _wtzset(void)
```

Description

Sets value of global variables _daylight, _timezone, and _tzname.

_tzset is available on XENIX systems.

_tzset sets the _daylight, _timezone, and _tzname global variables based on the environment variable TZ. _wtzset sets the _daylight, _timezone, and _wtzname global variables. The library functions ftime and localtime use these global variables to adjust Greenwich Mean Time (GMT) to the local time zone. The format of the TZ environment string is:

```
TZ = zzz[+/-]d[d][lll]
```

where zzz is a three-character string representing the name of the current time zone. All three characters are required. For example, the string “PST” could be used to represent Pacific standard time.

[+/-]d[d] is a required field containing an optionally signed number with 1 or more digits. This number is the local time zone’s

difference from GMT in hours. Positive numbers adjust westward from GMT. Negative numbers adjust eastward from GMT. For example, the number 5 = EST, +8 = PST, and -1 = continental Europe. This number is used in the calculation of the global variable _timezone. _timezone is the difference in seconds between GMT and the local time zone.

lll is an optional three-character field that represents the local time zone, daylight saving time. For example, the string “PDT” could be used to represent pacific daylight saving time. If this field is present, it causes the global variable _daylight to be set nonzero. If this field is absent, _daylight is set to zero.

If the TZ environment string isn’t present or isn’t in the preceding form, a default TZ = “EST5EDT” is presumed for the purposes of assigning values to the global variables _daylight, _timezone, and _tzname. On a Win32 system, none of these global variables are set if TZ is null.

The global variables _tzname[0] and _wtzname[1] point to a three-character string with the value of the time-zone name from the TZ environment string. _tzname[1] and _wtzname[1] point to a three-character string with the value of the daylight saving time-zone name from the TZ environment string. If no daylight saving name is present, _tzname[1] and _wtzname[1] point to a null string.

Return Value

None.

Example

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
   time_t td;
   putenv("TZ=PST8PDT");
   tzset();
   time(&td);
   printf("Current time = %s\n", asctime(localtime(&td)));
   return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_tzset	+	+		
_wtzset		+		

3.1.4.35 typeinfo.h

The following functions, macros, and classes are provided in `typeinfo.h`:

Topics

Name	Description
bad_cast class (see page 1212)	**Header File** typeinfo.h **Description** When dynamic_cast fails to make a cast to reference, the expression can throw bad_cast. Note that when dynamic_cast fails to make a cast to pointer type, the result is the null pointer.

bad_typeid class (see page 1212)	**Header File** typeinfo.h **Description** When the operand of typeid is a dereferenced null pointer, the **typeid** operator can throw bad_typeid.
type_info class (see page 1212)	**Header File** typeinfo.h **Description** Provides information about a type. **Constructor** Only a private constructor is provided. You cannot create type_info objects. By declaring your objects to be _ _rtti types, or by using the -RT compiler switch, the compiler provides your objects with the elements of type_info. type_info references are generated by the typeid operator. **Public Member Functions** **Operators**

3.1.4.35.1 bad_cast class

Header File

typeinfo.h

Description

When dynamic_cast fails to make a cast to reference, the expression can throw bad_cast. Note that when dynamic_cast fails to make a cast to pointer type, the result is the null pointer.

3.1.4.35.2 bad_typeid class

Header File

typeinfo.h

Description

When the operand of typeid is a dereferenced null pointer, the **typeid** operator can throw bad_typeid.

3.1.4.35.3 type_info class

Header File

typeinfo.h

Description

Provides information about a type.

Constructor

Only a private constructor is provided. You cannot create type_info objects. By declaring your objects to be _ _rtti types, or by using the -RT compiler switch, the compiler provides your objects with the elements of type_info. type_info references are generated by the typeid operator.

Public Member Functions

Operators

3.1.4.36 utime.h

The following functions, macros, and classes are provided in `utime.h`:

Topics

Name	Description

_utime, _wutime (see page 1213)	**Header File**
	utime.h
	Category
	Input/output Routines
	Prototype
	`int _utime(char *path, struct utimbuf *times);`
	`int _wutime(wchar_t *path, struct _utimbuf *times);`
	Description
	Sets file time and date.
	_utime sets the modification time for the file path. The modification time is contained in the utimbuf structure pointed to by times. This structure is defined in utime.h, and has the following format:
	`struct utimbuf {`
	`time_t actime; /* access time */`
	`time_t modtime; /* modification time */`
	`};`
	The FAT (file allocation table) file system supports only a modification time; therefore, on FAT file systems _utime ignores actime and uses only modtime to set... more (see page 1213)

3.1.4.36.1 _utime, _wutime

Header File

utime.h

Category

Input/output Routines

Prototype

```
int _utime(char *path, struct utimbuf *times);
int _wutime(wchar_t *path, struct _utimbuf *times);
```

Description

Sets file time and date.

_utime sets the modification time for the file path. The modification time is contained in the utimbuf structure pointed to by times. This structure is defined in utime.h, and has the following format:

```
struct utimbuf {
time_t actime; /* access time */
time_t modtime; /* modification time */
};
```

The FAT (file allocation table) file system supports only a modification time; therefore, on FAT file systems _utime ignores actime and uses only modtime to set the file's modification time.

If times is NULL, the file's modification time is set to the current time.

_wutime is the Unicode version of _utime. The Unicode version accepts a filename that is a wchar_t character string. Otherwise,

the functions perform identically.

Return Value

On success, _utime returns 0.

On error, it returns -1, and sets the global variable errno to one of the following values:

EACCES	Permission denied
EMFILE	Too many open files
ENOENT	Path or file name not found

Example

```
/* Copy timestamp from one file to another */
#include <sys\stat.h>
#include <utime.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
  struct stat src_stat;
  struct utimbuf times;
  if(argc != 3) {
    printf( "Usage: copytime <source file> <dest file>\n" );
    return 1;
  }

  if (stat(argv[1],&src_stat) != 0) {
    perror("Unable to get status of source file");
    return 1;
  }

  times.modtime = times.actime = src_stat.st_mtime;
  if (utime(argv[2],&times) != 0) {
    perror("Unable to set time of destination file");
    return 1;
  }
  return 0;
}
```

Portability

	POSIX	Win32	ANSI C	ANSI C++
_utime	+	+		
_wutime		+		

3.1.4.37 values.h

The following functions, macros, and classes are provided in `values.h`:

Topics

Name	Description
BITSPERBYTE #define (see page 1215)	**Header File** values.h **Description** Number of bits in a byte.

HIBITx #defines (see page 1215)	**Header File** values.h **Description** Bit mask for the high (sign) bit of standard integer types.
MAXxxxx #defines (integer data types) (see page 1215)	**Header File** values.h **Description** Maximum values for integer data types
Float and Double Limits (see page 1216)	**Header File** values.h **Description** **UNIX System V compatible:**

3.1.4.37.1 BITSPERBYTE #define

Header File

values.h

Description

Number of bits in a byte.

3.1.4.37.2 HIBITx #defines

Header File

values.h

Description

Bit mask for the high (sign) bit of standard integer types.

Name	Meaning
HIBITS	For type short
HIBITI	For type int
HIBITL	For type long

3.1.4.37.3 MAXxxxx #defines (integer data types)

Header File

values.h

Description

Maximum values for integer data types

Name	Meaning
MAXSHORT	Largest short
MAXINT	Largest int
MAXLONG	Largest long

3.1.4.37.4 Float and Double Limits

Header File

values.h

Description

UNIX System V compatible:

_LENBASE	Base to which exponent applies

Limits for double float values

_DEXPLEN	Number of bits in exponent
DMAXEXP	Maximum exponent allowed
DMAXPOWTWO	Largest power of two allowed
DMINEXP	Minimum exponent allowed
DSIGNIF	Number of significant bits
MAXDOUBLE	Largest magnitude double value
MINDOUBLE	Smallest magnitude double value

Limits for float values

_FEXPLEN	Number of bits in exponent
FMAXEXP	Maximum exponent allowed
FMAXPOWTWO	Largest power of two allowed
FMINEXP	Minimum exponent allowed
FSIGNIF	Number of significant bits
MAXFLOAT	Largest magnitude float value
MINFLOAT	Smallest magnitude float value

3.2 Win32 Developer's Guide

This section contains the Win32 Developer's Guide topics for the Delphi Win32 personality in RAD Studio.

Topics

Name	Description
Component Writer's Guide (see page 1217)	The Component Writer's Guide covers all the information relating to creating VCL components in the Delphi personality.
Developing COM-based Applications (see page 1381)	Contains the Developer's Guide topics for creating COM-based applications in Delphi.
Developing Database Applications (see page 1469)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress. Contains the Developer's Guide topics for programming database applications.
Programming with Delphi (see page 1879)	Contains the Developer's Guide topics for programming with Delphi.
Writing Internet Applications (see page 2243)	Contains the Developer's Guide topics for writing internet applications in Delphi.

3.2.1 Component Writer's Guide

The Component Writer's Guide covers all the information relating to creating VCL components in the Delphi personality.

Topics

Name	Description
Creating a graphic component (see page 1218)	
Creating events (see page 1231)	
Creating methods (see page 1241)	
Creating properties (see page 1245)	
Customizing a grid (see page 1258)	
Extending the IDE (see page 1276)	
Handling messages (see page 1298)	
Introduction to component creation (see page 1310)	
Making a control data aware (see page 1325)	
Making components available at design time (see page 1339)	
Making a dialog box a component (see page 1358)	
Modifying an existing component (see page 1363)	
Object-oriented programming for component writers (see page 1367)	
Using graphics in components (see page 1376)	

3.2.1.1 Creating a graphic component

Topics

Name	Description
Adding Graphic Capabilities (see page 1219)	Once you have declared your graphic component and published any inherited properties you want to make available, you can add the graphic capabilities that distinguish your component. You have two tasks to perform when creating a graphic control: 1. Determining what to draw. (see page 1223) 2. Drawing the component image. (see page 1224) In addition, for the shape control example, you will add some properties that enable application developers to customize the appearance of the shape at design time.
Creating a Graphic Component (see page 1220)	A graphic control is a simple kind of component. Because a purely graphic control never receives focus, it does not have or need its own window handle. Users can still manipulate the control with the mouse, but there is no keyboard interface. The graphic control presented in the following topics is *TShape*, the shape component on the Additional page of the Tool palette. Although the component created is identical to the standard shape component, you need to call it something different to avoid duplicate identifiers. The following topics use the name *TSampleShape* and show you all the steps involved... more (see page 1220)
Creating and Registering the Graphic Component (see page 1220)	You create every component in the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. This process is outlined in Creating a new component. (see page 1317)
Declaring the Access Properties (see page 1221)	You can provide access to the owned objects of a component by declaring properties of the type of the objects. That gives developers a way to access the objects at design time or runtime. Usually, the read part of the property just references the class field, but the write part calls a method that enables the component to react to changes in the owned object. To the shape control, add properties that provide access to the pen and brush fields. You will also declare methods for reacting to changes to the pen or brush.
Declaring the Class Fields (see page 1222)	Each class a component owns must have a class field declared for it in the component. The class field ensures that the component always has a pointer to the owned object so that it can destroy the class before destroying itself. In general, a component initializes owned objects in its constructor and destroys them in its destructor. Fields for owned objects are nearly always declared as private. If applications (or other components) need access to the owned objects, you can declare **published** or **public** properties for this purpose. Add fields for a pen and brush to the shape control:
Declaring the Property (see page 1222)	When you declare a property, you usually need to declare a private field to store the data for the property, then specify methods for reading and writing the property value. Often, you don't need to use a method to read the value, but can just point to the stored data instead. For the shape control, you will declare a field that holds the current shape, then declare a property that reads that field and writes to it through a method call. Add the following declarations to *TSampleShape*:
Declaring the Property Type (see page 1223)	When you declare a property of a user-defined type, you must declare the type first, before the class that includes the property. The most common sort of user-defined type for properties is enumerated. For the shape control, you need an enumerated type with an element for each kind of shape the control can draw. Add the following type definition above the shape control class's declaration.
Determining What to Draw (see page 1223)	A graphic control can change its appearance to reflect a dynamic condition, including user input. A graphic control that always looks the same should probably not be a component at all. If you want a static image, you can import the image instead of using a control. In general, the appearance of a graphic control depends on some combination of its properties. The gauge control, for example, has properties that determine its shape and orientation and whether it shows its progress numerically as well as graphically. Similarly, the shape control has a property that determines what kind of shape it... more (see page 1223)

Drawing the Component Image (see page 1224)	The essential element of a graphic control is the way it paints its image on the screen. The abstract type *TGraphicControl* defines a method called *Paint* that you override to paint the image you want on your control. The *Paint* method for the shape control needs to do several things: • Use the pen and brush selected by the user. • Use the selected shape. • Adjust coordinates so that squares and circles use the same width and height.
Initializing Owned Classes (see page 1225)	If you add classes to your component, the component's constructor must initialize them so that the user can interact with the objects at runtime. Similarly, the component's destructor must also destroy the owned objects before destroying the component itself.
Overriding the Constructor and Destructor (see page 1226)	To change default property values and initialize owned classes for your component, you must override the inherited constructor and destructor. In both cases, remember always to call the inherited method in your new constructor or destructor.
Publishing Inherited Properties (Graphic) (see page 1227)	Once you derive a component type, you can decide which of the properties and events declared in the protected parts of the ancestor class you want to surface in the new component. *TGraphicControl* already publishes all the properties that enable the component to function as a control, so all you need to publish is the ability to respond to mouse events and handle drag-and-drop. Publishing inherited properties and events is explained in Publishing inherited properties (see page 1250) and Making events visible. (see page 1238) Both processes involve redeclaring just the name of the properties in the published part of the class declaration. For the shape... more (see page 1227)
Publishing the Pen and Brush (see page 1228)	By default, a canvas has a thin black pen and a solid white brush. To let developers change the pen and brush, you must provide classes for them to manipulate at design time, then copy the classes into the canvas during painting. Classes such as an auxiliary pen or brush are called *owned classes* because the component owns them and is responsible for creating and destroying them. Managing owned classes requires: 1. Declaring the class fields. (see page 1222) 2. Declaring the access properties. (see page 1221) 3. Initializing owned classes. (see page 1225) 4. Setting owned classes' properties. (see page 1229)
Refining the Shape Drawing (see page 1228)	The standard shape control does one more thing that your sample shape control does not yet do: it handles squares and circles as well as rectangles and ellipses. To do that, you need to write code that finds the shortest side and centers the image. Here is a refined *Paint* method that adjusts for squares and ellipses:
Setting Owned Classes' Properties (see page 1229)	As the final step in handling the pen and brush classes, you need to make sure that changes in the pen and brush cause the shape control to repaint itself. Both pen and brush classes have *OnChange* events, so you can create a method in the shape control and point both *OnChange* events to it. Add the following method to the shape control, and update the component's constructor to set the pen and brush events to the new method:
Writing the Implementation Method (see page 1230)	When the **read** or **write** part of a property definition uses a method instead of directly accessing the stored property data, you need to implement the method. Add the implementation of the *SetShape* method to the **implementation** part of the unit:

3.2.1.1.1 Adding Graphic Capabilities

Once you have declared your graphic component and published any inherited properties you want to make available, you can add the graphic capabilities that distinguish your component. You have two tasks to perform when creating a graphic control:

1. Determining what to draw. (see page 1223)
2. Drawing the component image. (see page 1224)

In addition, for the shape control example, you will add some properties that enable application developers to customize the

appearance of the shape at design time.

3.2.1.1.2 Creating a Graphic Component

A graphic control is a simple kind of component. Because a purely graphic control never receives focus, it does not have or need its own window handle. Users can still manipulate the control with the mouse, but there is no keyboard interface.

The graphic control presented in the following topics is *TShape*, the shape component on the Additional page of the Tool palette. Although the component created is identical to the standard shape component, you need to call it something different to avoid duplicate identifiers. The following topics use the name *TSampleShape* and show you all the steps involved in creating the shape component:

- Creating and registering the component (see page 1363).
- Publishing inherited properties (see page 1227).
- Adding graphic capabilities (see page 1219).

3.2.1.1.3 Creating and Registering the Graphic Component

You create every component in the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. This process is outlined in Creating a new component. (see page 1317)

For this example, follow the general procedure for creating a component, with these specifics:

1. Call the component's unit *Shapes*.
2. Derive a new component type called *TSampleShape*, descended from *TGraphicControl*.
3. Register *TSampleShape* on the Samples category of the Tool palette.

The resulting unit should look like this:

```
unit Shapes;
interface
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TSampleShape = class(TGraphicControl)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponent('Samples', [TSampleShape]);
end;
end.
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Shapes.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//
static inline void ValidCtrCheck(TSampleShape *)
{
  new TSampleShape(NULL);
}
//---------------------------------------------------------------------------
__fastcall TSampleShape::TGraphicControl(TComponent* Owner)
: TGraphicControl(Owner)
```

```
{
}
//---------------------------------------------------------------------------
namespace Shapes
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleShape)};
    RegisterComponents("Samples", classes, 0);
  }
}
//---------------------------------------------------------------------------
#ifndef ShapesH
#define ShapesH
//---------------------------------------------------------------------------
#include <sysutils.hpp>
#include <controls.hpp>
#include <classes.hpp>
#include <forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleShape : public TGraphicControl
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

3.2.1.1.4 Declaring the Access Properties

You can provide access to the owned objects of a component by declaring properties of the type of the objects. That gives developers a way to access the objects at design time or runtime. Usually, the read part of the property just references the class field, but the write part calls a method that enables the component to react to changes in the owned object.

To the shape control, add properties that provide access to the pen and brush fields. You will also declare methods for reacting to changes to the pen or brush.

```
type
  TSampleShape = class(TGraphicControl)
  .
  .
  .
  private                                              { these methods should be private }
    procedure SetBrush(Value: TBrush);
    procedure SetPen(Value: TPen);
  published                                          { make these available at design time }
    property Brush: TBrush read FBrush write SetBrush;
    property Pen: TPen read FPen write SetPen;
  end;
class PACKAGE TSampleShape : public TGraphicControl
{
  .
  .
  .
private:
    TPen *FPen;
    TBrush *FBrush;
    void __fastcall SetBrush(TBrush *Value);
    void __fastcall SetPen(TPen *Value);
    .
    .
    .
__published:
```

```
    __property TBrush* Brush = {read=FBrush, write=SetBrush, nodefault};
    __property TPen* Pen = {read=FPen, write=SetPen, nodefault};
};
```

Then, write the *SetBrush* and *SetPen* methods in the implementation part of the unit:

```
procedure TSampleShape.SetBrush(Value: TBrush);
begin
  FBrush.Assign(Value);                          { replace existing brush with parameter }
end;
procedure TSampleShape.SetPen(Value: TPen);
begin
  FPen.Assign(Value);                             { replace existing pen with parameter }
end;
void __fastcall TSampleShape::SetBrush( TBrush* Value)
{
  FBrush->Assign(Value);
}
void __fastcall TSampleShape::SetPen( TPen* Value)
{
  FPen->Assign(Value);
}
```

To directly assign the contents of *Value* to *FBrush*-

```
  FBrush := Value;
  FBrush = Value;
```

- would overwrite the internal pointer for *FBrush*, lose memory, and create a number of ownership problems.

3.2.1.1.5 Declaring the Class Fields

Each class a component owns must have a class field declared for it in the component. The class field ensures that the component always has a pointer to the owned object so that it can destroy the class before destroying itself. In general, a component initializes owned objects in its constructor and destroys them in its destructor.

Fields for owned objects are nearly always declared as private. If applications (or other components) need access to the owned objects, you can declare **published** or **public** properties for this purpose.

Add fields for a pen and brush to the shape control:

```
type
  TSampleShape = class(TGraphicControl)
  private            { fields are nearly always private }
    FPen: TPen;      { a field for the pen object }
    FBrush: TBrush;  { a field for the brush object }
    .
    .
    .
  end;
class PACKAGE TSampleShape : public TGraphicControl
{
private:               // data members are always private
    TPen *FPen;        // a data member for the pen object
    TBrush *FBrush;    // a data member for the brush object
    .
    .
    .
};
```

3.2.1.1.6 Declaring the Property

When you declare a property, you usually need to declare a private field to store the data for the property, then specify methods for reading and writing the property value. Often, you don't need to use a method to read the value, but can just point to the

stored data instead.

For the shape control, you will declare a field that holds the current shape, then declare a property that reads that field and writes to it through a method call.

Add the following declarations to *TSampleShape*:

```
type
  TSampleShape = class(TGraphicControl)
  private
    FShape: TSampleShapeType;  { field to hold property value }
    procedure SetShape(Value: TSampleShapeType);
  published
    property Shape: TSampleShapeType read FShape write SetShape;
  end;
```

```
class PACKAGE TSampleShape : public TGraphicControl
{
private:
    TSampleShapeType FShape;
    void __fastcall SetShape(TSampleShapeType Value);
__published:
    __property TSampleShapeType Shape = {read=FShape, write=SetShape, nodefault};
};
```

Now all that remains is to add the implementation of *SetShape*.

3.2.1.1.7 Declaring the Property Type

When you declare a property of a user-defined type, you must declare the type first, before the class that includes the property. The most common sort of user-defined type for properties is enumerated.

For the shape control, you need an enumerated type with an element for each kind of shape the control can draw.

Add the following type definition above the shape control class's declaration.

```
type
  TSampleShapeType = (sstRectangle, sstSquare, sstRoundRect, sstRoundSquare,
    sstEllipse, sstCircle);
  TSampleShape = class(TGraphicControl) { this is already there }
```

```
enum TSampleShapeType { sstRectangle, sstSquare, sstRoundRect, sstRoundSquare, sstEllipse,
sstCircle };
class PACKAGE TSampleShape : public TGraphicControl         // this is already there
```

You can now use this type to declare a new property in the class.

3.2.1.1.8 Determining What to Draw

A graphic control can change its appearance to reflect a dynamic condition, including user input. A graphic control that always looks the same should probably not be a component at all. If you want a static image, you can import the image instead of using a control.

In general, the appearance of a graphic control depends on some combination of its properties. The gauge control, for example, has properties that determine its shape and orientation and whether it shows its progress numerically as well as graphically. Similarly, the shape control has a property that determines what kind of shape it should draw.

To give your control a property that determines the shape it draws, add a property called *Shape*. This requires

1. Declaring the property type. (see page 1223)
2. Declaring the property. (see page 1222)
3. Writing the implementation method. (see page 1230)

Creating properties is explained in more detail in Creating properties (see page 1249).

3.2.1.1.9 Drawing the Component Image

The essential element of a graphic control is the way it paints its image on the screen. The abstract type *TGraphicControl* defines a method called *Paint* that you override to paint the image you want on your control.

The *Paint* method for the shape control needs to do several things:

- Use the pen and brush selected by the user.
- Use the selected shape.
- Adjust coordinates so that squares and circles use the same width and height.

Overriding the Paint method requires two steps:

1. Add *Paint* to the component's declaration.
2. Write the *Paint* method in the **implementation** part of the unit.

For the shape control, add the following declaration to the class declaration:

```
type
  TSampleShape = class(TGraphicControl)
  .
  .
  .
  protected
    procedure Paint; override;
  .
  .
  .
  end;
class PACKAGE TSampleShape : public TGraphicControl
{
  .
  .
  .
protected:
    virtual void __fastcall Paint();
    .
    .
    .
};
```

Then write the method in the **implementation** part of the unit:

```
procedure TSampleShape.Paint;
begin
  with Canvas do
  begin
    Pen := FPen;                                          { copy the component's pen }
    Brush := FBrush;                                    { copy the component's brush }
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(0, 0, Width, Height);                 { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(0, 0, Width, Height, Width div 4, Height div 4); { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(0, 0, Width, Height);                         { draw round shapes }
    end;
  end;
end;
void __fastcall TSampleShape::Paint()
{
  int X,Y,W,H,S;
```

```
  Canvas->Pen = FPen;                                   // copy the component's pen
  Canvas->Brush = FBrush;                               // copy the component's brush
  W=Width;                                              // use the component width
  H=Height;                                             // use the component height
  X=Y=0;                                                // save smallest for circles/squares
  if( W<H )
    S=W;
  else
    S=H;
  switch(FShape)
  {
    case sstRectangle:                                   // draw rectangles and squares
    case sstSquare:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundRect:                                   // draw rounded rectangles and squares
    case sstRoundSquare:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                                      // draw circles and ellipses
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

Paint is called whenever the control needs to update its image. Controls are painted when they first appear or when a window in front of them goes away. In addition, you can force repainting by calling *Invalidate*, as the *StyleChanged* method does.

3.2.1.1.10 Initializing Owned Classes

If you add classes to your component, the component's constructor must initialize them so that the user can interact with the objects at runtime. Similarly, the component's destructor must also destroy the owned objects before destroying the component itself.

Because you have added a pen and a brush to the shape control, you need to initialize them in the shape control's constructor and destroy them in the control's destructor:

1. Construct the pen and brush in the shape control constructor:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                        { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                                  { construct the pen }
  FBrush := TBrush.Create;                                            { construct the brush }
end;
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();                             // construct the pen
  FPen = new TPen();                                 // construct the brush
}
```

2. Add the overridden destructor to the declaration of the component class:

```
type
  TSampleShape = class(TGraphicControl)
  public                                                  { destructors are always public}
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;                          { remember override directive }
```

```
  end;
class PACKAGE TSampleShape : public TGraphicControl
{
  .
  .
  .
public:                                                  // destructors are always public
    virtual __fastcall TSampleShape(TComponent* Owner);
    __fastcall ~TSampleShape();                          // the destructor
    .
    .
    .
};
```

3. Write the new destructor in the **implementation** part of the unit:

```
destructor TSampleShape.Destroy;
begin
  FPen.Free;                                                  { destroy the pen object }
  FBrush.Free;                                               { destroy the brush object }
  inherited Destroy;                          { always call the inherited destructor, too }
end;
__fastcall TSampleShape::~TSampleShape()
{
  delete FPen;                                           // delete the pen object
  delete FBrush;                                         // delete the brush object
}
```

3.2.1.1.11 Overriding the Constructor and Destructor

To change default property values and initialize owned classes for your component, you must override the inherited constructor and destructor. In both cases, remember always to call the inherited method in your new constructor or destructor.

Changing default property values

The default size of a graphic control is fairly small, so you can change the width and height in the constructor. Changing default property values is explained in more detail in Modifying an existing component (see page 1365).

In this example, the shape control sets its size to a square 65 pixels on each side.

1. Add the overridden constructor to the declaration of the component class:

```
type
  TSampleShape = class(TGraphicControl)
  public                                                  { constructors are always public }
    constructor Create(AOwner: TComponent); override        { remember override directive }
  end;
class PACKAGE TSampleShape : public TGraphicControl
{
public:
    virtual __fastcall TSampleShape(TComponent *Owner);
};
```

2. Redeclare the *Height* and *Width* properties with their new default values:

```
type
  TSampleShape = class(TGraphicControl)
  .
  .
  .
  published
    property Height default 65;
    property Width default 65;
  end;
class PACKAGE TSampleShape : public TGraphicControl
{
  .
```

```
  .
  .
__published:
    __property Height;
    __property Width;
}
```

3. Write the new constructor in the **implementation** part of the unit:

```
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);  { always call the inherited constructor }
  Width := 65;
  Height := 65;
end;
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
}
```

3.2.1.1.12 Publishing Inherited Properties (Graphic)

Once you derive a component type, you can decide which of the properties and events declared in the protected parts of the ancestor class you want to surface in the new component. *TGraphicControl* already publishes all the properties that enable the component to function as a control, so all you need to publish is the ability to respond to mouse events and handle drag-and-drop.

Publishing inherited properties and events is explained in Publishing inherited properties (see page 1250) and Making events visible. (see page 1238) Both processes involve redeclaring just the name of the properties in the published part of the class declaration.

For the shape control, you can publish the three mouse events, the three drag-and-drop events, and the two drag-and-drop properties:

```
type
  TSampleShape = class(TGraphicControl)
  published
    property DragCursor;        { drag-and-drop properties }
    property DragMode;
    property OnDragDrop;        { drag-and-drop events }
    property OnDragOver;
    property OnEndDrag;
    property OnMouseDown;       { mouse events }
    property OnMouseMove;
    property OnMouseUp;
  end;
class PACKAGE TSampleShape : public TGraphicControl
{
private:
__published:
    __property DragCursor ;
    __property DragMode ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnMouseDown ;
    __property OnMouseMove ;
    __property OnMouseUp ;
};
```

The sample shape control now makes mouse and drag-and-drop interactions available to its users.

3.2.1.1.13 Publishing the Pen and Brush

By default, a canvas has a thin black pen and a solid white brush. To let developers change the pen and brush, you must provide classes for them to manipulate at design time, then copy the classes into the canvas during painting. Classes such as an auxiliary pen or brush are called *owned classes* because the component owns them and is responsible for creating and destroying them.

Managing owned classes requires:

1. Declaring the class fields. (see page 1222)
2. Declaring the access properties. (see page 1221)
3. Initializing owned classes. (see page 1225)
4. Setting owned classes' properties. (see page 1229)

3.2.1.1.14 Refining the Shape Drawing

The standard shape control does one more thing that your sample shape control does not yet do: it handles squares and circles as well as rectangles and ellipses. To do that, you need to write code that finds the shortest side and centers the image.

Here is a refined *Paint* method that adjusts for squares and ellipses:

```
procedure TSampleShape.Paint;
var
  X, Y, W, H, S: Integer;
begin  with Canvas do
  begin
    Pen := FPen;                                          { copy the component's pen }
    Brush := FBrush;                                    { copy the component's brush }
    W := Width;                                            { use the component width }
    H := Height;                                         { use the component height }
    if W < H then S := W else S := H;                 { save smallest for circles/squares }
    case FShape of                                    { adjust height, width and position }
      sstRectangle, sstRoundRect, sstEllipse:
        begin
          X := 0;                                  { origin is top-left for these shapes }
          Y := 0;
        end;
      sstSquare, sstRoundSquare, sstCircle:
        begin
          X := (W - S) div 2;                             { center these horizontally... }
          Y := (H - S) div 2;                                       { ...and vertically }
          W := S;                                  { use shortest dimension for width... }
          H := S;                                               { ...and for height }
        end;
    end;
    case FShape of
      sstRectangle, sstSquare:
        Rectangle(X, Y, X + W, Y + H);                    { draw rectangles and squares }
      sstRoundRect, sstRoundSquare:
        RoundRect(X, Y, X + W, Y + H, S div 4, S div 4);           { draw rounded shapes }
      sstCircle, sstEllipse:
        Ellipse(X, Y, X + W, Y + H);                                 { draw round shapes }
    end;
  end;
end;
void __fastcall TSampleShape::Paint(void)
{
  int X,Y,W,H,S;
  Canvas->Pen = FPen;                        // copy the component's pen
  Canvas->Brush = FBrush;                    // copy the component's brush
```

```
  W=Width;                                    // use the component width
  H=Height;                                   // use the component height
  X=Y=0;                                      // save smallest for circles/squares
  if( W<H )
    S=W;
  else
    S=H;
  switch(FShape)                               // adjust height, width and position
  {
    case sstRectangle:
    case sstRoundRect:
    case sstEllipse:
      Y=X=0;                                 // origin is top-left for these shapes
      break;
    case sstSquare:
    case sstRoundSquare:
    case sstCircle:
      X= (W-S)/2;                             // center these horizontally
      Y= (H-S)/2;                             // and vertically
      break;
    default:
    break;
  }
  switch(FShape)
  {
    case sstSquare:                           // draw rectangles and squares
      W=H=S;                                  // use shortest dimension for width and height
    case sstRectangle:
      Canvas->Rectangle(X,Y,X+W,Y+H);
      break;
    case sstRoundSquare:                      // draw rounded rectangles and squares
      W=H=S;
    case sstRoundRect:
      Canvas->RoundRect(X,Y,X+W,Y+H,S/4,S/4);
      break;
    case sstCircle:                           // draw circles and ellipses
      W=H=S;
    case sstEllipse:
      Canvas->Ellipse(X,Y,X+W,Y+H);
      break;
    default:
      break;
  }
}
```

3.2.1.1.15 Setting Owned Classes' Properties

As the final step in handling the pen and brush classes, you need to make sure that changes in the pen and brush cause the shape control to repaint itself. Both pen and brush classes have *OnChange* events, so you can create a method in the shape control and point both *OnChange* events to it.

Add the following method to the shape control, and update the component's constructor to set the pen and brush events to the new method:

```
type
  TSampleShape = class(TGraphicControl)
  published
    procedure StyleChanged(Sender: TObject);
  end;
.
.
.
implementation
.
.
```

```
.
constructor TSampleShape.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                      { always call the inherited constructor }
  Width := 65;
  Height := 65;
  FPen := TPen.Create;                                                { construct the pen }
  FPen.OnChange := StyleChanged;                    { assign method to OnChange event }
  FBrush := TBrush.Create;                                        { construct the brush }
  FBrush.OnChange := StyleChanged;                  { assign method to OnChange event }
end;
procedure TSampleShape.StyleChanged(Sender: TObject);
begin
  Invalidate;                                    { erase and repaint the component }
end;
//header file
class PACKAGE TSampleShape : public TGraphicControl
{
  .
  .
  .
public:
    void __fastcall StyleChanged(TObject* Owner);
    .
    .
    .
};
//implmentation file
__fastcall TSampleShape::TSampleShape(TComponent* Owner) : TGraphicControl(Owner)
{
  Width = 65;
  Height = 65;
  FBrush = new TBrush();
  FBrush->OnChange = StyleChanged;
  FPen = new TPen();
  FPen->OnChange = StyleChanged;
}
//also include StyleChanged method in the implementation file
void __fastcall TSampleShape::StyleChanged( TObject* Sender)
{
  Invalidate();         // repaints the component
}
```

With these changes, the component redraws to reflect changes to either the pen or the brush.

3.2.1.1.16 Writing the Implementation Method

When the **read** or **write** part of a property definition uses a method instead of directly accessing the stored property data, you need to implement the method.

Add the implementation of the *SetShape* method to the **implementation** part of the unit:

```
procedure TSampleShape.SetShape(Value: TSampleShapeType);
begin
  if FShape <> Value then                          { ignore if this isn"t a change }
  begin
    FShape := Value;                               { store the new value }
    Invalidate;                                    { force a repaint with the new shape }
  end;
end;
void __fastcall TSampleShape::SetShape(TSampleShapeType Value)
{
  if (FShape != Value)             // ignore if this isn't a change
  {
     FShape = Value;               // store the new value
```

```
      Invalidate();                // force a repaint with the new shape
    }
  }
```

3.2.1.2 Creating events

Topics

Name	Description
Creating Events: Overview (see page 1233)	An event is a link between an occurrence in the system (such as a user action or a change in focus) and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the application developer. Events let application developers customize the behavior of components without having to change the classes themselves. This is known as *delegation*. Events for the most common user actions (such as mouse actions) are built into all the standard components, but you can also define new events. To create events in a... more (see page 1233)
What Are Events? (see page 1234)	An event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer that points to a method in a specific class instance. From the application developer's perspective, an event is just a name related to a system occurrence, such as *OnClick*, to which specific code can be attached. For example, a push button called *Button1* has an *OnClick* method. By default, when you assign a value to the *OnClick* event, the Form Designer generates an event handler called *Button1Click* in the form that contains the button and assigns it to *OnClick... more (see page 1234)*
Events Are closures (C++) (see page 1234)	Closures are used to implement events. A closure is a special pointer type that points to a specific method in a specific class instance. As a component writer, you can treat the closure as a place holder: your code detects that an event occurs, so you call the method (if any) specified by the user for that event. Closures maintain a hidden pointer to a class instance. When the user assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a specific method of a specific class instance.... more (see page 1234)
Events Are Method Pointers (see page 1234)	Delphi uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in a specific class instance. As a component writer, you can treat the method pointer as a placeholder: When your code detects that an event occurs, you call the method (if any) specified by the user for that event. Method pointers work just like any other procedural type, but they maintain a hidden pointer to a class instance. When the application developer assigns a handler to a component's event, the assignment is not just to a method with a... more (see page 1234)
Calling the Click-event Handler (see page 1235)	All controls, for example, inherit a dynamic method called *Click* for handling click events:
Events Are Properties (see page 1235)	Components use properties to implement their events. Unlike most other properties, events do not use methods to implement their read and write parts. Instead, event properties use a private class field of the same type as the property. By convention, the field's name is the name of the property preceded by the letter *F*. For example, the *OnClick* method's pointer is stored in a field called *FOnClick* of type *TNotifyEvent*, and the declaration of the *OnClick* event property looks like this:
Event Types Are Method-pointer Types (see page 1235)	Because an event is a pointer to an event handler, the type of the event property must be a method-pointer type. Similarly, any code to be used as an event handler must be an appropriately typed method of a class. All event-handler methods are procedures. To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way. Delphi defines method types for all its standard events. When you create your own events, you can use an existing type if that is... more (see page 1235)

Event Handler Types Are Procedures (see page 1236)	Although the compiler allows you to declare method-pointer types that are functions, you should never do so for handling events. Because an empty function returns an undefined result, an empty event handler that was a function might not always be valid. For this reason, all your events and their associated event handlers should be procedures. Although an event handler cannot be a function, you can still get information from the application developer's code using **var** parameters. When doing this, make sure you assign a valid value to the parameter before calling the handler so you don't require the user's code... more (see page 1236)
Event Handlers Have A Return Type of void (C++) (see page 1236)	Event handlers must have a return type of **void**. Even though the handler can return only **void**, you can still get information back from the user's code by passing arguments by reference. When you do this, make sure you assign a valid value to the argument before calling the handler so you do not require the user's code to change the value. An example of passing arguments by reference to an event handler is the key-pressed event, of type `TKeyPressEvent`. `TKeyPressEvent` defines two arguments: one to indicate which object generated the event, and one to indicate which... more (see page 1236)
Event Types Are closure Types (C++) (see page 1236)	Because an event is a pointer to an event handler, the type of the event property must be a closure type. Similarly, any code to be used as an event handler must be an appropriately typed method of a class. To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way. C++Builder defines closures for all its standard events. When you create your own events, you can use an existing closure if that is appropriate, or define one of your... more (see page 1236)
Event Handlers Are Optional (see page 1237)	When creating events, remember that developers using your components may not attach handlers to them. This means that your component should not fail or generate errors simply because there is no handler attached to a particular event. (The mechanics of calling handlers and dealing with events that have no attached handler are explained in Calling the event (see page 1240).) Events happen almost constantly in a GUI application. Just moving the mouse pointer across a visual component sends numerous mouse-move messages, which the component translates into *OnMouseMove* events. In most cases, developers do not want to handle the mouse-move events, and this... more (see page 1237)
Implementing the Standard Events (see page 1237)	The controls that come with the component library inherit events for the most common occurrences. These are called the *standard events*. Although all these events are built into the controls, they are often **protected**, meaning developers cannot attach handlers to them. When you create a control, you can choose to make events visible to users of your control. There are three things you need to consider when incorporating the standard events into your controls: • Identifying standard events (see page 1237) • Making events visible (see page 1238) • Changing the standard event handling (see page 1238)
Identifying Standard Events (see page 1237)	There are two categories of standard events: those defined for all controls and those defined only for the standard windowed controls.
Making Events Visible (see page 1238)	The declarations of the standard events in *TControl* and *TWinControl* are protected, as are the methods that correspond to them. If you are inheriting from one of these abstract classes and want to make their events accessible at runtime or design time, you need to redeclare the events as either **public** or **published**. Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. You can, therefore, take an event that is defined in *TControl* but not made visible, and surface it by declaring it as public or published.
Changing the Standard Event Handling (see page 1238)	If you want to change the way your component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As an application developer, that is exactly what you would do. But when you are creating a component, you must keep the event available for developers who use the component. This is the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling; and by calling the inherited method you can maintain the standard handling,... more (see page 1238)

Defining Your Own Events (see page 1238)	Defining entirely new events is relatively unusual. There are times, however, when a component introduces behavior that is entirely different from that of any other component, so you will need to define an event for it. There are the issues you will need to consider when defining an event: • Triggering the event (see page 1239) • Defining the handler type (see page 1239) • Declaring the event (see page 1240) • Calling the event (see page 1240)
Triggering the Event (see page 1239)	You need to know what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse and Windows sends a *WM_LBUTTONDOWN* message to the application. Upon receiving that message, a component calls its *MouseDown* method, which in turn calls any code the user has attached to the *OnMouseDown* event. However, some events are less clearly tied to specific external occurrences. For example, a scroll bar has an *OnChange* event, which is triggered by several kinds of occurrence, including keystrokes, mouse clicks, and changes in other... more (see page 1239)
Two Kinds of Events (see page 1239)	There are two kinds of occurrence you might need to provide events for: user interactions and state changes. User-interaction events are nearly always triggered by a message from Windows, indicating that the user did something your component may need to respond to. State-change events may also be related to messages from Windows (focus changes or enabling, for example), but they can also occur through changes in properties or other code. You have total control over the triggering of the events you define. Define the events with care so that developers are able to understand and use them.
Defining the Handler Type (see page 1239)	Once you determine when the event occurs, you must define how you want the event handled. This means determining the type of the event handler. In most cases, handlers for events you define yourself are either simple notifications or event-specific types. It is also possible to get information back from the handler.
Declaring the Event (see page 1240)	Once you have determined the type of your event handler, you are ready to declare the method pointer and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.
Calling the Event (see page 1240)	You should centralize calls to an event. That is, create a virtual method in your component that calls the application's event handler (if it assigns one) and provides any default handling. Putting all the event calls in one place ensures that someone deriving a new component from yours can customize event handling by overriding a single method, rather than searching through your code for places where you call the event. There are two other considerations when calling the event: • Empty handlers must be valid. (see page 1240) • Users can override default handling. (see page 1240)
Empty Handlers Must Be Valid (see page 1240)	You should never create a situation in which an empty event handler causes an error, nor should the proper functioning of your component depend on a particular response from the application's event-handling code.
Users Can Override Default Handling (see page 1240)	For some kinds of events, developers may want to replace the default handling or even suppress all responses. To allow this, you need to pass an argument by reference to the handler and check for a certain value when the handler returns. This is in keeping with the rule that an empty handler should have the same effect as no handler at all. Because an empty handler will not change the values of arguments passed by reference, the default handling always takes place after calling the empty handler.

3.2.1.2.1 Creating Events: Overview

An event is a link between an occurrence in the system (such as a user action or a change in focus) and a piece of code that responds to that occurrence. The responding code is an *event handler*, and is nearly always written by the application developer. Events let application developers customize the behavior of components without having to change the classes themselves. This is known as *delegation*.

Events for the most common user actions (such as mouse actions) are built into all the standard components, but you can also define new events. To create events in a component, you need to understand the following:

- What are events? (see page 1234)
- Implementing the standard events (see page 1237)
- Defining your own events (see page 1238)

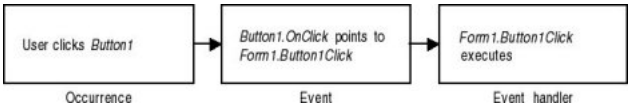
Events are implemented as properties, so you should already be familiar with the material in Creating properties (see page 1249) before you attempt to create or change a component's events.

3.2.1.2.2 What Are Events?

An event is a mechanism that links an occurrence to some code. More specifically, an event is a method pointer that points to a method in a specific class instance.

From the application developer's perspective, an event is just a name related to a system occurrence, such as *OnClick*, to which specific code can be attached. For example, a push button called *Button1* has an *OnClick* method. By default, when you assign a value to the *OnClick* event, the Form Designer generates an event handler called *Button1Click* in the form that contains the button and assigns it to *OnClick*. When a click event occurs in the button, the button calls the method assigned to *OnClick*, in this case, *Button1Click*.

To write an event, you need to understand the following:



- Events are method pointers (see page 1234).
- Events are properties (see page 1235).
- Event types are method-pointer types (see page 1235).
- Event-handler types are procedures (see page 1236).
- Event handlers are optional (see page 1237).

3.2.1.2.3 Events Are closures (C++)

Closures are used to implement events. A closure is a special pointer type that points to a specific method in a specific class instance. As a component writer, you can treat the closure as a place holder: your code detects that an event occurs, so you call the method (if any) specified by the user for that event.

Closures maintain a hidden pointer to a class instance. When the user assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a specific method of a specific class instance. That instance is usually the form that contains the component, but it need not be.

3.2.1.2.4 Events Are Method Pointers

Delphi uses method pointers to implement events. A method pointer is a special pointer type that points to a specific method in a specific class instance. As a component writer, you can treat the method pointer as a placeholder: When your code detects that an event occurs, you call the method (if any) specified by the user for that event.

Method pointers work just like any other procedural type, but they maintain a hidden pointer to a class instance. When the application developer assigns a handler to a component's event, the assignment is not just to a method with a particular name, but rather to a method of a specific class instance. That instance is usually the form that contains the component, but it need not be.

3.2.1.2.5 Calling the Click-event Handler

All controls, for example, inherit a dynamic method called *Click* for handling click events:

```
procedure Click; dynamic;
virtual void __fastcall Click(void);
```

The implementation of *Click* calls the user's click-event handler, if one exists. If the user has assigned a handler to a control's *OnClick* event, clicking the control results in that method being called. If no handler is assigned, nothing happens.

3.2.1.2.6 Events Are Properties

Components use properties to implement their events. Unlike most other properties, events do not use methods to implement their read and write parts. Instead, event properties use a private class field of the same type as the property.

By convention, the field's name is the name of the property preceded by the letter *F*. For example, the *OnClick* method's pointer is stored in a field called *FOnClick* of type *TNotifyEvent*, and the declaration of the *OnClick* event property looks like this:

```
type
  TControl = class(TComponent)
  private
    FOnClick: TNotifyEvent;                { declare a field to hold the method pointer }
    .
    .
    .
  protected
    property OnClick: TNotifyEvent read FOnClick write FOnClick;
  end;
class PACKAGE TControl : public TComponent
{
private:
    TNotifyEvent FOnClick;
  .
  .
  .
protected:
    __property TNotifyEvent OnClick = {read=FOnClick, write=FOnClick};
  .
  .
  .
};
```

To learn about *TNotifyEvent* and other event types, see the next section, Event types are method-pointer types. (see page 1235)

As with any other property, you can set or change the value of an event at runtime. The main advantage to having events be properties, however, is that component users can assign handlers to events at design time, using the Object Inspector.

3.2.1.2.7 Event Types Are Method-pointer Types

Because an event is a pointer to an event handler, the type of the event property must be a method-pointer type. Similarly, any code to be used as an event handler must be an appropriately typed method of a class.

All event-handler methods are procedures. To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

Delphi defines method types for all its standard events. When you create your own events, you can use an existing type if that is appropriate, or define one of your own.

3.2.1.2.8 Event Handler Types Are Procedures

Although the compiler allows you to declare method-pointer types that are functions, you should never do so for handling events. Because an empty function returns an undefined result, an empty event handler that was a function might not always be valid. For this reason, all your events and their associated event handlers should be procedures.

Although an event handler cannot be a function, you can still get information from the application developer's code using **var** parameters. When doing this, make sure you assign a valid value to the parameter before calling the handler so you don't require the user's code to change the value.

An example of passing **var** parameters to an event handler is the *OnKeyPress* event, of type *TKeyPressEvent*. *TKeyPressEvent* defines two parameters, one to indicate which object generated the event, and one to indicate which key was pressed:

```
type
  TKeyPressEvent = procedure(Sender: TObject; var Key: Char) of object;
```

Normally, the *Key* parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component may want to change the character. One example might be to force all characters to uppercase in an editor. In that case, the user could define the following handler for keystrokes:

```
procedure TForm1.Edit1KeyPressed(Sender: TObject; var Key: Char);
begin
  Key := UpCase(Key);
end;
```

You can also use **var** parameters to let the user override the default handling.

3.2.1.2.9 Event Handlers Have A Return Type of void (C++)

Event handlers must have a return type of **void**. Even though the handler can return only **void**, you can still get information back from the user's code by passing arguments by reference. When you do this, make sure you assign a valid value to the argument before calling the handler so you do not require the user's code to change the value.

An example of passing arguments by reference to an event handler is the key-pressed event, of type `TKeyPressEvent`. `TKeyPressEvent` defines two arguments: one to indicate which object generated the event, and one to indicate which key was pressed:

```
typedef void __fastcall (__closure *TKeyPressEvent)(TObject *Sender, Char &Key);
```

Normally, the `Key` parameter contains the character pressed by the user. Under certain circumstances, however, the user of the component might want to change the character. One example might be to force all characters to uppercase in an edit control. In that case, the user could define the following handler for keystrokes:

```
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, Char &Key)
{
  Key = UpCase(Key);
}
```

You can also use arguments passed by reference to let the user override the default handling.

3.2.1.2.10 Event Types Are closure Types (C++)

Because an event is a pointer to an event handler, the type of the event property must be a closure type. Similarly, any code to be used as an event handler must be an appropriately typed method of a class.

To be compatible with an event of a given type, an event-handler method must have the same number and type of parameters, in the same order, passed in the same way.

C++Builder defines closures for all its standard events. When you create your own events, you can use an existing closure if that is appropriate, or define one of your own.

3.2.1.2.11 Event Handlers Are Optional

When creating events, remember that developers using your components may not attach handlers to them. This means that your component should not fail or generate errors simply because there is no handler attached to a particular event. (The mechanics of calling handlers and dealing with events that have no attached handler are explained in Calling the event (see page 1240).)

Events happen almost constantly in a GUI application. Just moving the mouse pointer across a visual component sends numerous mouse-move messages, which the component translates into *OnMouseMove* events. In most cases, developers do not want to handle the mouse-move events, and this should not cause a problem. So the components you create should not require handlers for their events.

Moreover, application developers can write any code they want in an event handler. The components in the class library have events written in such a way as to minimize the chance of an event handler generating errors. Obviously, you cannot protect against logic errors in application code, but you can ensure that data structures are initialized before calling events so that application developers do not try to access invalid data.

3.2.1.2.12 Implementing the Standard Events

The controls that come with the component library inherit events for the most common occurrences. These are called the *standard events*. Although all these events are built into the controls, they are often **protected**, meaning developers cannot attach handlers to them. When you create a control, you can choose to make events visible to users of your control.

There are three things you need to consider when incorporating the standard events into your controls:

- Identifying standard events (see page 1237)
- Making events visible (see page 1238)
- Changing the standard event handling (see page 1238)

3.2.1.2.13 Identifying Standard Events

There are two categories of standard events: those defined for all controls and those defined only for the standard windowed controls.

Standard events for all controls

The most basic events are defined in the class *TControl*. All controls, whether windowed, graphical, or custom, inherit these events. The following events are available in all controls:

- *OnClick*
- *OnDblClick*
- *OnDragDrop*
- *OnDragOver*
- *OnEndDrag*
- *OnMouseMove*
- *OnMouseDown*
- *OnMouseUp*

The standard events have corresponding protected virtual methods declared in *TControl*, with names that correspond to the event names. For example, *OnClick* events call a method named *Click*, and *OnEndDrag* events call a method named

DoEndDrag.

Standard events for standard controls

In addition to the events common to all controls, standard windowed controls (those that descend from *TWinControl*) have the following events:

- *OnEnter*
- *OnKeyPress*
- *OnKeyDown*
- *OnKeyUp*
- *OnExit*

Like the standard events in *TControl*, the windowed control events have corresponding methods. The standard key events listed above respond to all normal keystrokes.

Note: To respond to special keystrokes (such as the Alt

key), however, you must respond to the WM_GETDLGCODE or CM_WANTSPECIALKEYS message from Windows. See Handling messages and system notifications (see page 1308) for information on writing message handlers.

3.2.1.2.14 Making Events Visible

The declarations of the standard events in *TControl* and *TWinControl* are protected, as are the methods that correspond to them. If you are inheriting from one of these abstract classes and want to make their events accessible at runtime or design time, you need to redeclare the events as either **public** or **published**.

Redeclaring a property without specifying its implementation keeps the same implementation methods, but changes the protection level. You can, therefore, take an event that is defined in *TControl* but not made visible, and surface it by declaring it as public or published.

3.2.1.2.15 Changing the Standard Event Handling

If you want to change the way your component responds to a certain kind of event, you might be tempted to write some code and assign it to the event. As an application developer, that is exactly what you would do. But when you are creating a component, you must keep the event available for developers who use the component.

This is the reason for the protected implementation methods associated with each of the standard events. By overriding the implementation method, you can modify the internal event handling; and by calling the inherited method you can maintain the standard handling, including the event for the application developer's code.

The order in which you call the methods is significant. As a rule, call the inherited method first, allowing the application developer's event-handler to execute before your customizations (and in some cases, to keep the customizations from executing). There may be times when you want to execute your code before calling the inherited method, however. For example, if the inherited code is somehow dependent on the status of the component and your code changes that status, you should make the changes and then allow the user's code to respond to them.

3.2.1.2.16 Defining Your Own Events

Defining entirely new events is relatively unusual. There are times, however, when a component introduces behavior that is entirely different from that of any other component, so you will need to define an event for it.

There are the issues you will need to consider when defining an event:

- Triggering the event (see page 1239)

- Defining the handler type (see page 1239)
- Declaring the event (see page 1240)
- Calling the event (see page 1240)

3.2.1.2.17 Triggering the Event

You need to know what triggers the event. For some events, the answer is obvious. For example, a mouse-down event occurs when the user presses the left button on the mouse and Windows sends a *WM_LBUTTONDOWN* message to the application. Upon receiving that message, a component calls its *MouseDown* method, which in turn calls any code the user has attached to the *OnMouseDown* event.

However, some events are less clearly tied to specific external occurrences. For example, a scroll bar has an *OnChange* event, which is triggered by several kinds of occurrence, including keystrokes, mouse clicks, and changes in other controls. When defining your events, you must ensure that all the appropriate occurrences call the proper events.

3.2.1.2.18 Two Kinds of Events

There are two kinds of occurrence you might need to provide events for: user interactions and state changes. User-interaction events are nearly always triggered by a message from Windows, indicating that the user did something your component may need to respond to. State-change events may also be related to messages from Windows (focus changes or enabling, for example), but they can also occur through changes in properties or other code.

You have total control over the triggering of the events you define. Define the events with care so that developers are able to understand and use them.

3.2.1.2.19 Defining the Handler Type

Once you determine when the event occurs, you must define how you want the event handled. This means determining the type of the event handler. In most cases, handlers for events you define yourself are either simple notifications or event-specific types. It is also possible to get information back from the handler.

Simple notifications

A notification event is one that only tells you that the particular event happened, with no specific information about when or where. Notifications use the type *TNotifyEvent*, which carries only one parameter, the sender of the event. All a handler for a notification "knows" about the event is what kind of event it was, and what component the event happened to. For example, click events are notifications. When you write a handler for a click event, all you know is that a click occurred and which component was clicked.

Notification is a one-way process. There is no mechanism to provide feedback or prevent further handling of a notification.

Event-specific handlers

In some cases, it is not enough to know which event happened and what component it happened to. For example, if the event is a key-press event, it is likely that the handler will want to know which key the user pressed. In these cases, you need handler types that include parameters for additional information.

If your event was generated in response to a message, it is likely that the parameters you pass to the event handler come directly from the message parameters.

Returning information from the handler

Because all event handlers are procedures, the only way to pass information back from a handler is through a **var** parameter. Your components can use such information to determine how or whether to process an event after the user's handler executes.

For example, all the key events (*OnKeyDown*, *OnKeyUp*, and *OnKeyPress*) pass by reference the value of the key pressed in a parameter named *Key*. The event handler can change *Key* so that the application sees a different key as being involved in the event. This is a way to force typed characters to uppercase, for example.

3.2.1.2.20 Declaring the Event

Once you have determined the type of your event handler, you are ready to declare the method pointer and the property for the event. Be sure to give the event a meaningful and descriptive name so that users can understand what the event does. Try to be consistent with names of similar properties in other components.

Event names start with "On"

The names of most events in Delphi begin with "On." This is just a convention; the compiler does not enforce it. The Object Inspector determines that a property is an event by looking at the type of the property: all method-pointer properties are assumed to be events and appear on the Events page.

Developers expect to find events in the alphabetical list of names starting with "On." Using other kinds of names is likely to confuse them.

Note: The main exception to this rule is that many events that occur before and after some occurrence begin with "Before" and "After."

3.2.1.2.21 Calling the Event

You should centralize calls to an event. That is, create a virtual method in your component that calls the application's event handler (if it assigns one) and provides any default handling.

Putting all the event calls in one place ensures that someone deriving a new component from yours can customize event handling by overriding a single method, rather than searching through your code for places where you call the event.

There are two other considerations when calling the event:

- Empty handlers must be valid. (see page 1240)
- Users can override default handling. (see page 1240)

3.2.1.2.22 Empty Handlers Must Be Valid

You should never create a situation in which an empty event handler causes an error, nor should the proper functioning of your component depend on a particular response from the application's event-handling code.

3.2.1.2.23 Users Can Override Default Handling

For some kinds of events, developers may want to replace the default handling or even suppress all responses. To allow this, you need to pass an argument by reference to the handler and check for a certain value when the handler returns.

This is in keeping with the rule that an empty handler should have the same effect as no handler at all. Because an empty handler will not change the values of arguments passed by reference, the default handling always takes place after calling the empty handler.

3.2.1.3 Creating methods

Topics

Name	Description
Abstract Methods (see page 1242)	Sometimes a method is declared as **abstract** in a Delphi component. In the component library, abstract methods usually occur in classes whose names begin with "custom," such as *TCustomGrid*. Such classes are themselves abstract, in the sense that they are intended only for deriving descendant classes. While you can create an instance object of a class that contains an abstract member, it is not recommended. Calling the abstract member leads to an *EAbstractError* exception. The **abstract** directive is used to indicate parts of classes that should be surfaced and defined in descendant components; it forces component writers to redeclare... more (see page 1242)
Avoiding Interdependencies (see page 1242)	At all times when writing components, minimize the preconditions imposed on the developer. To the greatest extent possible, developers should be able to do anything they want to a component, whenever they want to do it. There will be times when you cannot accommodate that, but your goal should be to come as close as possible. This list gives you an idea of the kinds of dependencies to avoid: • Methods that the user *must* call to use the component • Methods that must execute in a particular order • Methods that put the component into a state or mode where certain events... more (see page 1242)
Creating Methods: Overview (see page 1243)	Component methods are procedures and functions built into the structure of a class. Although there are essentially no restrictions on what you can do with the methods of a component, Delphi does use some standards you should follow. These guidelines include: • Avoiding dependencies (see page 1242) • Naming methods (see page 1244) • Protecting methods (see page 1245) • Making methods virtual (see page 1245) • Declaring methods (see page 1243) In general, components should not contain many methods and you should minimize the number of methods that an application needs to call. The features you might be inclined to implement as methods are often better encapsulated into properties. Properties provide an interface that suits the Delphi and are... more (see page 1243)
Declaring Methods (see page 1243)	Declaring a method in a component is the same as declaring any class method. To declare a new method in a component, do the following: • Add the declaration to the component's object-type declaration. • Implement the method in the **implementation** part of the component's unit.
Example of Declaring Methods (see page 1243)	The following code shows a component that defines two new methods, one **protected** method and one **public** virtual method.

Methods That Should Be Protected (see page 1244)	Any implementation methods for the component should be **protected** so that applications cannot call them at the wrong time. If you have methods that application code should not call, but that are called in derived classes, declare them as **protected**. For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method **public**, there is a chance that applications will call it before setting up the data. On the other hand, by making it **protected**, you ensure that applications cannot call it directly. You can then... more (see page 1244)
Methods That Should Be Public (see page 1244)	Any method that application developers need to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid tying up system resources or putting the operating system in a state where it cannot respond to the user. **Note:** Constructors and destructors should always be **public**.
Naming Methods (see page 1244)	Delphi imposes no restrictions on what you name methods or their parameters. There are a few conventions that make methods easier for application developers, however. Keep in mind that the nature of a component architecture dictates that many different kinds of people can use your components. If you are accustomed to writing code that only you or a small group of programmers use, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use... more (see page 1244)
Protecting Methods (see page 1245)	All parts of classes, including fields, methods, and properties, have a level of protection or "visibility," as explained in Controlling access. (see page 1371) Choosing the appropriate visibility for a method is simple. Most methods you write in your components are **public** or **protected**. You rarely need to make a method **private**, unless it is truly specific to that type of component, to the point that even derived components should not have access to it.
Making Methods Virtual (see page 1245)	You make methods **virtual** when you want different types to be able to execute different code in response to the same method call. If you create components intended to be used directly by application developers, you can probably make all your methods nonvirtual. On the other hand, if you create abstract components from which other components will be derived, consider making the added methods **virtual**. This way, derived components can override the inherited **virtual** methods.

3.2.1.3.1 Abstract Methods

Sometimes a method is declared as **abstract** in a Delphi component. In the component library, abstract methods usually occur in classes whose names begin with "custom," such as *TCustomGrid*. Such classes are themselves abstract, in the sense that they are intended only for deriving descendant classes.

While you can create an instance object of a class that contains an abstract member, it is not recommended. Calling the abstract member leads to an *EAbstractError* exception.

The **abstract** directive is used to indicate parts of classes that should be surfaced and defined in descendant components; it forces component writers to redeclare the abstract member in descendant classes before actual instances of the class can be created.

3.2.1.3.2 Avoiding Interdependencies

At all times when writing components, minimize the preconditions imposed on the developer. To the greatest extent possible, developers should be able to do anything they want to a component, whenever they want to do it. There will be times when you cannot accommodate that, but your goal should be to come as close as possible.

This list gives you an idea of the kinds of dependencies to avoid:

- Methods that the user *must* call to use the component
- Methods that must execute in a particular order
- Methods that put the component into a state or mode where certain events or methods could be invalid

The best way to handle these situations is to ensure that you provide ways out of them. For example, if calling a method puts your component into a state where calling another method might be invalid, then write that second method so that if an application calls it when the component is in a bad state, the method corrects the state before executing its main code. At a minimum, you should raise an exception in cases when a user calls a method that is invalid.

In other words, if you create a situation where parts of your code depend on each other, the burden should be on *you* to be sure that using the code in incorrect ways does not cause problems. A warning message, for example, is preferable to a system failure if the user does not accommodate your dependencies.

3.2.1.3.3 Creating Methods: Overview

Component methods are procedures and functions built into the structure of a class. Although there are essentially no restrictions on what you can do with the methods of a component, Delphi does use some standards you should follow. These guidelines include:

- Avoiding dependencies (see page 1242)
- Naming methods (see page 1244)
- Protecting methods (see page 1245)
- Making methods virtual (see page 1245)
- Declaring methods (see page 1243)

In general, components should not contain many methods and you should minimize the number of methods that an application needs to call. The features you might be inclined to implement as methods are often better encapsulated into properties. Properties provide an interface that suits the Delphi and are accessible at design time.

3.2.1.3.4 Declaring Methods

Declaring a method in a component is the same as declaring any class method.

To declare a new method in a component, do the following:

- Add the declaration to the component's object-type declaration.
- Implement the method in the **implementation** part of the component's unit.

3.2.1.3.5 Example of Declaring Methods

The following code shows a component that defines two new methods, one **protected** method and one **public** virtual method.

C++

This is the interface definition in the `.H` file:

```
class PACKAGE TSampleComponent : public TControl
{
protected:
    void __fastcall MakeBigger();
public:
    virtual int __fastcall CalculateArea();
  .
  .
  .
};
```

This is the code in the .CPP file of the unit that implements the methods:

```
void __fastcall TSampleComponent::MakeBigger()
{
  Height = Height + 5;
  Width = Width + 5;
```

```
}
int __fastcall TSampleComponent::CalculateArea()
{
  return Width * Height;
}
```

Delphi

```
type
  TSampleComponent = class(TControl)
  protected
    procedure MakeBigger;                                { declare protected static method }
  public
    function CalculateArea: Integer; virtual;            { declare public virtual method }
  end;
.
.
.
implementation
.
.
.
procedure TSampleComponent.MakeBigger;                          { implement first method }
begin
  Height := Height + 5;
  Width := Width + 5;
end;
function TSampleComponent.CalculateArea: Integer;              { implement second method }
begin
  Result := Width * Height;
end;
```

3.2.1.3.6 Methods That Should Be Protected

Any implementation methods for the component should be **protected** so that applications cannot call them at the wrong time. If you have methods that application code should not call, but that are called in derived classes, declare them as **protected**.

For example, suppose you have a method that relies on having certain data set up for it beforehand. If you make that method **public**, there is a chance that applications will call it before setting up the data. On the other hand, by making it **protected**, you ensure that applications cannot call it directly. You can then set up other, **public** methods that ensure that data setup occurs before calling the **protected** method.

Property-implementation methods should be declared as virtual **protected** methods. Methods that are so declared allow the application developers to override the property implementation, either augmenting its functionality or replacing it completely. Such properties are fully polymorphic. Keeping access methods **protected** ensures that developers do not accidentally call them, inadvertently modifying a property.

3.2.1.3.7 Methods That Should Be Public

Any method that application developers need to call must be declared as **public**. Keep in mind that most method calls occur in event handlers, so methods should avoid tying up system resources or putting the operating system in a state where it cannot respond to the user.

Note: Constructors and destructors should always be **public**.

3.2.1.3.8 Naming Methods

Delphi imposes no restrictions on what you name methods or their parameters. There are a few conventions that make methods easier for application developers, however. Keep in mind that the nature of a component architecture dictates that many different

kinds of people can use your components.

If you are accustomed to writing code that only you or a small group of programmers use, you might not think too much about how you name things. It is a good idea to make your method names clear because people unfamiliar with your code (and even unfamiliar with coding) might have to use your components.

Here are some suggestions for making clear method names:

- Make names descriptive. Use meaningful verbs. A name like *PasteFromClipboard* is much more informative than simply *Paste* or *PFC*.
- Function names should reflect the nature of what they return.

Although it might be obvious to you as a programmer that a function named *X* returns the horizontal position of something, a name like *GetHorizontalPosition* is more universally understandable.

As a final consideration, make sure the method really needs to be a method. A good guideline is that method names have verbs in them. If you find that you create a lot of methods that do not have verbs in their names, consider whether those methods ought to be properties.

3.2.1.3.9 Protecting Methods

All parts of classes, including fields, methods, and properties, have a level of protection or "visibility," as explained in Controlling access. (see page 1371) Choosing the appropriate visibility for a method is simple.

Most methods you write in your components are **public** or **protected**. You rarely need to make a method **private**, unless it is truly specific to that type of component, to the point that even derived components should not have access to it.

3.2.1.3.10 Making Methods Virtual

You make methods **virtual** when you want different types to be able to execute different code in response to the same method call.

If you create components intended to be used directly by application developers, you can probably make all your methods nonvirtual. On the other hand, if you create abstract components from which other components will be derived, consider making the added methods **virtual**. This way, derived components can override the inherited **virtual** methods.

3.2.1.4 Creating properties

Topics

Name	Description
Creating Properties: Overview (see page 1249)	Properties are the most visible parts of components. The application developer can see and manipulate them at design time and get immediate feedback as the components react in the Form Designer. Well-designed properties make your components easier for others to use and easier for you to maintain. To make the best use of properties in your components, you should understand the following: • Why create properties? (see page 1249) • Types of properties (see page 1250) • Publishing inherited properties (see page 1250) • Defining properties (see page 1251) • Creating array properties (see page 1253) • Storing and loading properties (see page 1254)

Why Create Properties? (see page 1249)	From the application developer's standpoint, properties look like variables. Developers can set or read the values of properties as if they were fields. (About the only thing you can do with a variable that you cannot do with a property is pass it as a **var** parameter.) Properties provide more power than simple fields because • Application developers can set properties at design time. Unlike methods, which are available only at runtime, properties let the developer customize components before running an application. Properties can appear in the Object Inspector, which simplifies the programmer's job; instead of handling several parameters to construct... more (see page 1249)
Types of Properties (see page 1250)	A property can be of any type. Different types are displayed differently in the Object Inspector, which validates property assignments as they are made at design time. ***How properties appear in the Object Inspector***
Publishing Inherited Properties (see page 1250)	All components inherit properties from their ancestor classes. When you derive a new component from an existing one, your new component inherits all the properties of its immediate ancestor. If you derive from one of the abstract classes, many of the inherited properties are either protected or public, but not published. To make a protected or public property available at design time in the Object Inspector, you must redeclare the property as published. Redeclaring means adding a declaration for the inherited property to the declaration of the descendant class.
Defining Properties (see page 1251)	This section shows how to declare new properties and explains some of the conventions followed in the standard components. Topics include: • Property declarations (see page 1251) • Internal data storage (see page 1251) • Direct access (see page 1251) • Access methods (see page 1252) • Default property values (see page 1253)
Property Declarations (see page 1251)	A property is declared in the declaration of its component class. To declare a property, you specify three things: • The name of the property. • The type of the property. • The methods used to read and write the value of the property. If no write method is declared, the property is read-only. Properties declared in a **published** section of the component's class declaration are editable in the Object Inspector at design time. The value of a published property is saved with the component in the form file. Properties declared in a **public** section are available at runtime and can be read... more (see page 1251)

Internal Data Storage (see page 1251)	There are no restrictions on how you store the data for a property. In general, however, Delphi components follow these conventions: • Property data is stored in class fields. • The fields used to store property data are private and should be accessed only from within the component itself. Derived components should use the inherited property; they do not need direct access to the property's internal data storage. • Identifiers for these fields consist of the letter *F* followed by the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in a field called... more (see page 1251)
Direct Access (see page 1251)	The simplest way to make property data available is *direct access*. That is, the **read** and **write** parts of the property declaration specify that assigning or reading the property value goes directly to the internal-storage field without calling an access method. Direct access is useful when you want to make a property available in the Object Inspector but changes to its value trigger no immediate processing. It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part. This allows the status of the component to be... more (see page 1251)
Access Methods (properties) (see page 1252)	You can specify an access method instead of a field in the **read** and **write** parts of a property declaration. Access methods should be protected, and are usually declared as **virtual**; this allows descendant components to override the property's implementation. Avoid making access methods public. Keeping them protected ensures that application developers do not inadvertently modify a property by calling one of these methods.
The Read Method (see page 1252)	The read method for a property is a function that takes no parameters (except as noted below) and returns a value of the same type as the property. By convention, the function's name is *Get* followed by the name of the property. For example, the read method for a property called *Count* would be *GetCount*. The read method manipulates the internal storage data as needed to produce the value of the property in the appropriate type. The only exceptions to the no-parameters rule are for array properties and properties that use index specifiers (see Creating array properties (see page 1253)), both... more (see page 1252)
The Write Method (see page 1252)	The write method for a property is a procedure that takes a single parameter (except as noted below) of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the write method's name is *Set* followed by the name of the property. For example, the write method for a property called *Count* would be *SetCount*. The value passed in the parameter becomes the new value of the property; the write method must perform any manipulation needed to put the appropriate data in the property's... more (see page 1252)
Default Property Values (see page 1253)	When you declare a property, you can specify a *default value* for it. The VCL uses the default value to determine whether to store the property in a form file. If you do not specify a default value for a property, the VCL always stores the property. To specify a default value for a property, append the **default** directive to the property's declaration (or redeclaration), followed by the default value. For example,
Specifying No Default Value (see page 1253)	When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one. To designate a property as having no default value, append the **nodefault** directive to the property's declaration. For example,

Creating Array Properties (see page 1253)	Some properties lend themselves to being indexed like arrays. For example, the *Lines* property of *TMemo* is an indexed list of the strings that make up the text of the memo; you can treat it as an array of strings. *Lines* provides natural access to a particular element (a string) in a larger set of data (the memo text). Array properties are declared like other properties, except that • The declaration includes one or more indexes with specified types. The indexes can be of any type. • The **read** and **write** parts of the property declaration, if specified, must be methods. They... more (see page 1253)
Creating Properties for Interfaces (see page 1253)	You can use an interface as the value of a published property, much as you can use an object. However, the mechanism by which your component receives notifications from the implementation of that interface differs. In Creating properties for subcomponents (see page 1254), the property setter called the *FreeNotification* method of the component that was assigned as the property value. This allowed the component to update itself when the component that was the value of the property was freed. When the value of the property is an interface, however, you don't have access to the component that implements that interface. As a... more (see page 1253)
Creating Properties for Subcomponents (see page 1254)	By default, when a property's value is another component, you assign a value to that property by adding an instance of the other component to the form or data module and then assigning that component as the value of the property. However, it is also possible for your component to create its own instance of the object that implements the property value. Such a dedicated component is called a subcomponent. Subcomponents can be any persistent object (any descendant of *TPersistent*). Unlike separate components that happen to be assigned as the value of a property, the published properties of subcomponents... more (see page 1254)
Storing and Loading Properties (see page 1254)	Delphi stores forms and their components in form (.dfm in VCL applications) files. A form file stores the properties of a form and its components. When Delphi developers add the components you write to their forms, your components must have the ability to write their properties to the form file when saved. Similarly, when loaded into Delphi or executed as part of an application, the components must restore themselves from the form file. Most of the time you will not need to do anything to make your components work with form files because the ability to store a representation... more (see page 1254)
Using the Store-and-load Mechanism (see page 1255)	The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description. By default, when storing itself, a component writes the values of all its published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, setting all properties to their default values, then reads the stored, non-default property values. This default mechanism serves the needs of most components, and requires no action... more (see page 1255)
Specifying Default Values (see page 1255)	Delphi components save their property values only if those values differ from the defaults. If you do not specify otherwise, Delphi assumes a property has no default value, meaning the component always stores the property, whatever its value. To specify a default value for a property, add the **default** directive and the new default value to the end of the property declaration. You can also specify a default value when re-declaring a property. In fact, one reason to re-declare a property is to designate a different default value.
Determining What to Store (see page 1255)	You can control whether Delphi stores each of your components' properties. By default, all properties in the published part of the class declaration are stored. You can choose not to store a given property at all, or you can designate a function that determines dynamically whether to store the property. To control whether Delphi stores a property, add the **stored** directive to the property declaration, followed by *True*, *False*, or the name of a Boolean function.

Initializing After Loading (see page 1256)	After a component reads all its property values from its stored description, it calls a virtual method named *Loaded*, which performs any required initializations. The call to *Loaded* occurs before the form and its controls are shown, so you do not need to worry about initialization causing flicker on the screen. To initialize a component after it loads its property values, override the *Loaded* method. **Note:** The first thing to do in any *Loaded* method is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you initialize your own component. The following code... more (see page 1256)
Storing and Loading Unpublished Properties (see page 1256)	By default, only published properties are loaded and saved with a component. However, it is possible to load and save unpublished properties. This allows you to have persistent properties that do not appear in the Object Inspector. It also allows components to store and load property values that Delphi does not know how to read or write because the value of the property is too complex. For example, the *TStrings* object can't rely on Delphi's automatic behavior to store and load the strings it represents and must use the following mechanism. You can save unpublished properties by adding code that... more (see page 1256)
Creating Methods to Store and Load Property Values (see page 1256)	To store and load unpublished properties, you must first create a method to store your property value and another to load your property value. You have two choices: • Create a method of type *TWriterProc* to store your property value and a method of type *TReaderProc* to load your property value. This approach lets you take advantage of Delphi's built-in capabilities for saving and loading simple types. If your property value is built out of types that Delphi knows how to save and load, use this approach. • Create two methods of type *TStreamProc*, one to store and one to load... more (see page 1256)
Overriding the DefineProperties Method (see page 1257)	Once you have created methods to store and load your property value, you can override the component's *DefineProperties* method. Delphi calls this method when it loads or stores the component. In the *DefineProperties* method, you must call the *DefineProperty* method or the *DefineBinaryProperty* method of the current filer, passing it the method to use for loading or saving your property value. If your load and store methods are of type *TWriterProc* and type *TReaderProc*, then you call the filer's *DefineProperty* method. If you created methods of type *TStreamProc*, call *DefineBinaryProperty* instead. No matter which method you use to... more (see page 1257)

3.2.1.4.1 Creating Properties: Overview

Properties are the most visible parts of components. The application developer can see and manipulate them at design time and get immediate feedback as the components react in the Form Designer. Well-designed properties make your components easier for others to use and easier for you to maintain.

To make the best use of properties in your components, you should understand the following:

- Why create properties? (see page 1249)
- Types of properties (see page 1250)
- Publishing inherited properties (see page 1250)
- Defining properties (see page 1251)
- Creating array properties (see page 1253)
- Storing and loading properties (see page 1254)

3.2.1.4.2 Why Create Properties?

From the application developer's standpoint, properties look like variables. Developers can set or read the values of properties

as if they were fields. (About the only thing you can do with a variable that you cannot do with a property is pass it as a **var** parameter.)

Properties provide more power than simple fields because

- Application developers can set properties at design time. Unlike methods, which are available only at runtime, properties let the developer customize components before running an application. Properties can appear in the Object Inspector, which simplifies the programmer's job; instead of handling several parameters to construct an object, the Object Inspector supplies the values. The Object Inspector also validates property assignments as soon as they are made.
- Properties can hide implementation details. For example, data stored internally in an encrypted form can appear unencrypted as the value of a property; although the value is a simple number, the component may look up the value in a database or perform complex calculations to arrive at it. Properties let you attach complex effects to outwardly simple assignments; what looks like an assignment to a field can be a call to a method which implements elaborate processing.
- Properties can be virtual. Hence, what looks like a single property to an application developer may be implemented differently in different components.

A simple example is the *Top* property of all controls. Assigning a new value to *Top* does not just change a stored value; it repositions and repaints the control. And the effects of setting a property need not be limited to an individual component; for example, setting the *Down* property of a speed button to *True* sets *Down* property of all other speed buttons in its group to *False*.

3.2.1.4.3 Types of Properties

A property can be of any type. Different types are displayed differently in the Object Inspector, which validates property assignments as they are made at design time.

How properties appear in the Object Inspector

Property type	treatment
Simple	Numeric, character, and string properties appear as numbers, characters, and strings. The application developer can edit the value of the property directly.
Enumerated	Properties of enumerated types (including Boolean) appear as editable strings. The developer can also cycle through the possible values by double-clicking the value column, and there is a drop-down list that shows all possible values.
Set	Properties of set types appear as sets. By double-clicking on the property, the developer can expand the set and treat each element as a Boolean value (true if it is included in the set).
Object	Properties that are themselves classes often have their own property editors, specified in the component's registration procedure. If the class held by a property has its own published properties, the Object Inspector lets the developer to expand the list (by double-clicking) to include these properties and edit them individually. Object properties must descend from *TPersistent*.
Interface	Properties that are interfaces can appear in the Object Inspector as long as the value is an interface that is implemented by a component (a descendant of *TComponent*). Interface properties often have their own property editors.
Array	Array properties must have their own property editors; the Object Inspector has no built-in support for editing them. You can specify a property editor when you register your components.

3.2.1.4.4 Publishing Inherited Properties

All components inherit properties from their ancestor classes. When you derive a new component from an existing one, your new component inherits all the properties of its immediate ancestor. If you derive from one of the abstract classes, many of the inherited properties are either protected or public, but not published.

To make a protected or public property available at design time in the Object Inspector, you must redeclare the property as

published. Redeclaring means adding a declaration for the inherited property to the declaration of the descendant class.

3.2.1.4.5 Defining Properties

This section shows how to declare new properties and explains some of the conventions followed in the standard components. Topics include:

- Property declarations (see page 1251)
- Internal data storage (see page 1251)
- Direct access (see page 1251)
- Access methods (see page 1252)
- Default property values (see page 1253)

3.2.1.4.6 Property Declarations

A property is declared in the declaration of its component class. To declare a property, you specify three things:

- The name of the property.
- The type of the property.
- The methods used to read and write the value of the property. If no write method is declared, the property is read-only.

Properties declared in a **published** section of the component's class declaration are editable in the Object Inspector at design time. The value of a published property is saved with the component in the form file. Properties declared in a **public** section are available at runtime and can be read or set in program code.

3.2.1.4.7 Internal Data Storage

There are no restrictions on how you store the data for a property. In general, however, Delphi components follow these conventions:

- Property data is stored in class fields.
- The fields used to store property data are private and should be accessed only from within the component itself. Derived components should use the inherited property; they do not need direct access to the property's internal data storage.
- Identifiers for these fields consist of the letter *F* followed by the name of the property. For example, the raw data for the *Width* property defined in *TControl* is stored in a field called *FWidth*.

The principle that underlies these conventions is that only the implementation methods for a property should access the data behind it. If a method or another property needs to change that data, it should do so through the property, not by direct access to the stored data. This ensures that the implementation of an inherited property can change without invalidating derived components.

3.2.1.4.8 Direct Access

The simplest way to make property data available is *direct access*. That is, the **read** and **write** parts of the property declaration specify that assigning or reading the property value goes directly to the internal-storage field without calling an access method. Direct access is useful when you want to make a property available in the Object Inspector but changes to its value trigger no immediate processing.

It is common to have direct access for the **read** part of a property declaration but use an access method for the **write** part. This allows the status of the component to be updated when the property value changes.

3.2.1.4.9 Access Methods (properties)

You can specify an access method instead of a field in the **read** and **write** parts of a property declaration. Access methods should be protected, and are usually declared as **virtual**; this allows descendant components to override the property's implementation.

Avoid making access methods public. Keeping them protected ensures that application developers do not inadvertently modify a property by calling one of these methods.

3.2.1.4.10 The Read Method

The read method for a property is a function that takes no parameters (except as noted below) and returns a value of the same type as the property. By convention, the function's name is *Get* followed by the name of the property. For example, the read method for a property called *Count* would be *GetCount*. The read method manipulates the internal storage data as needed to produce the value of the property in the appropriate type.

The only exceptions to the no-parameters rule are for array properties and properties that use index specifiers (see Creating array properties (see page 1253)), both of which pass their index values as parameters. (Use index specifiers to create a single read method that is shared by several properties. For more information about index specifiers, see the *Delphi Language Guide*.)

If you do not declare a read method, the property is write-only. Write-only properties are seldom used.

3.2.1.4.11 The Write Method

The write method for a property is a procedure that takes a single parameter (except as noted below) of the same type as the property. The parameter can be passed by reference or by value, and can have any name you choose. By convention, the write method's name is *Set* followed by the name of the property. For example, the write method for a property called *Count* would be *SetCount*. The value passed in the parameter becomes the new value of the property; the write method must perform any manipulation needed to put the appropriate data in the property's internal storage.

The only exceptions to the single-parameter rule are for array properties and properties that use index specifiers, both of which pass their index values as a second parameter. (Use index specifiers to create a single write method that is shared by several properties. For more information about index specifiers, see the *Delphi Language Guide*.)

If you do not declare a write method, the property is read-only.

Write methods commonly test whether a new value differs from the current value before changing the property. For example, here is a simple write method for an integer property called *Count* that stores its current value in a field called *FCount*.

```
procedure TMyComponent.SetCount(Value: Integer);
begin
  if Value <> FCount then
  begin
    FCount := Value;
    Update;
  end;
end;
```

```
void __fastcall TMyComponent::SetCount( int Value )
{
  if ( Value != FCount )
  {
    FCount = Value;
    Update();
  }
```

3.2.1.4.12 Default Property Values

When you declare a property, you can specify a *default value* for it. The VCL uses the default value to determine whether to store the property in a form file. If you do not specify a default value for a property, the VCL always stores the property.

To specify a default value for a property, append the **default** directive to the property's declaration (or redeclaration), followed by the default value. For example,

```
property Cool Boolean read GetCool write SetCool default True;
__property bool IsTrue = {read=GetIsTrue, write=SetIsTrue, default=true};
```

Note: Declaring a default value does not set the property to that value. The component's constructor method should initialize property values when appropriate. However, since objects always initialize their fields to 0, it is not strictly necessary for the constructor to set integer properties to 0, string properties to null, or Boolean properties to *False*.

3.2.1.4.13 Specifying No Default Value

When redeclaring a property, you can specify that the property has no default value, even if the inherited property specified one.

To designate a property as having no default value, append the **nodefault** directive to the property's declaration. For example,

```
property FavoriteFlavor string nodefault;
__property int NewInteger = {nodefault};
```

When you declare a property for the first time, there is no need to include **nodefault**. The absence of a declared default value means that there is no default.

3.2.1.4.14 Creating Array Properties

Some properties lend themselves to being indexed like arrays. For example, the *Lines* property of *TMemo* is an indexed list of the strings that make up the text of the memo; you can treat it as an array of strings. *Lines* provides natural access to a particular element (a string) in a larger set of data (the memo text).

Array properties are declared like other properties, except that

- The declaration includes one or more indexes with specified types. The indexes can be of any type.
- The **read** and **write** parts of the property declaration, if specified, must be methods. They cannot be fields.

The read and write methods for an array property take additional parameters that correspond to the indexes. The parameters must be in the same order and of the same type as the indexes specified in the declaration.

There are a few important differences between array properties and arrays. Unlike the index of an array, the index of an array property does not have to be an integer type. You can index a property on a string, for example. In addition, you can reference only individual elements of an array property, not the entire range of the property.

3.2.1.4.15 Creating Properties for Interfaces

You can use an interface as the value of a published property, much as you can use an object. However, the mechanism by which your component receives notifications from the implementation of that interface differs. In Creating properties for subcomponents (see page 1254), the property setter called the *FreeNotification* method of the component that was assigned as the property value. This allowed the component to update itself when the component that was the value of the property was freed. When the value of the property is an interface, however, you don't have access to the component that implements that interface. As a result, you can't call its *FreeNotification* method.

To handle this situation, you can call your component's *ReferenceInterface* method:

```
procedure TDemoComponent.SetMyIntfProp(const Value: IMyInterface);
```

```
begin
  ReferenceInterface(FIntfField, opRemove);
  FIntfField := Value;
  ReferenceInterface(FIntfField, opInsert);
end;
```

Calling *ReferenceInterface* with a specified interface does the same thing as calling another component's *FreeNotification* method. Thus, after calling *ReferenceInterface* from the property setter, you can override the *Notification* method to handle the notifications from the implementor of the interface:

```
procedure TDemoComponent.Notification(AComponent: TComponent; Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Assigned(MyIntfProp)) and (AComponent.IsImplementorOf(MyInftProp)) then
    MyIntfProp := nil;
end;
```

Note that the *Notification* code assigns **nil** to the *MyIntfProp* property, not to the private field (*FIntfField*). This ensures that *Notification* calls the property setter, which calls *ReferenceInterface* to remove the notification request that was established when the property value was set previously. All assignments to the interface property must be made through the property setter.

3.2.1.4.16 Creating Properties for Subcomponents

By default, when a property's value is another component, you assign a value to that property by adding an instance of the other component to the form or data module and then assigning that component as the value of the property. However, it is also possible for your component to create its own instance of the object that implements the property value. Such a dedicated component is called a subcomponent.

Subcomponents can be any persistent object (any descendant of *TPersistent*). Unlike separate components that happen to be assigned as the value of a property, the published properties of subcomponents are saved with the component that creates them. In order for this to work, however, the following conditions must be met:

- The *Owner* of the subcomponent must be the component that creates it and uses it as the value of a published property. For subcomponents that are descendants of *TComponent*, you can accomplish this by setting the *Owner* property of the subcomponent. For other subcomponents, you must override the *GetOwner* method of the persistent object so that it returns the creating component.
- If the subcomponent is a descendant of *TComponent*, it must indicate that it is a subcomponent by calling the *SetSubComponent* method. Typically, this call is made either by the owner when it creates the subcomponent or by the constructor of the subcomponent.

  Note: When a component that has subcomponents is streamed, the subcomponents will have their csLoading flag set and their Loaded method called. This can create a complication for any subcomponent properties that are writable. If you allow your subcomponent property to be assigned to an external component reference, then you cannot free your subcomponent until it's owner's Loaded method is called. Otherwise, the streaming system will attempt to call the subcomponent's Loaded method after the subcomponent has been freed.

  Typically, properties whose values are subcomponents are read-only. If you allow a property whose value is a subcomponent to be changed, the property setter must free the subcomponent when another component is assigned as the property value. In addition, the component often re-instantiates its subcomponent when the property is set to nil. Otherwise, once the property is changed to another component, the subcomponent can never be restored at design time.

  Note that the property setter above called the *FreeNotification* method of the component that is set as the property value. This call ensures that the component that is the value of the property sends a notification if it is about to be destroyed. It sends this notification by calling the *Notification* method. You handle this call by overriding the *Notification* method.

3.2.1.4.17 Storing and Loading Properties

Delphi stores forms and their components in form (.dfm in VCL applications) files. A form file stores the properties of a form and its components. When Delphi developers add the components you write to their forms, your components must have the ability to

write their properties to the form file when saved. Similarly, when loaded into Delphi or executed as part of an application, the components must restore themselves from the form file.

Most of the time you will not need to do anything to make your components work with form files because the ability to store a representation and load from it are part of the inherited behavior of components. Sometimes, however, you might want to alter the way a component stores itself or the way it initializes when loaded; so you should understand the underlying mechanism.

These are the aspects of property storage you need to understand:

- Using the store-and-load mechanism (see page 1255)
- Specifying default values (see page 1255)
- Determining what to store (see page 1255)
- Initializing after loading (see page 1256)
- Storing and loading unpublished properties (see page 1256)

3.2.1.4.18 Using the Store-and-load Mechanism

The description of a form consists of a list of the form's properties, along with similar descriptions of each component on the form. Each component, including the form itself, is responsible for storing and loading its own description.

By default, when storing itself, a component writes the values of all its published properties that differ from their default values, in the order of their declaration. When loading itself, a component first constructs itself, setting all properties to their default values, then reads the stored, non-default property values.

This default mechanism serves the needs of most components, and requires no action at all on the part of the component writer. There are several ways you can customize the storing and loading process to suit the needs of your particular components, however.

3.2.1.4.19 Specifying Default Values

Delphi components save their property values only if those values differ from the defaults. If you do not specify otherwise, Delphi assumes a property has no default value, meaning the component always stores the property, whatever its value.

To specify a default value for a property, add the **default** directive and the new default value to the end of the property declaration.

You can also specify a default value when re-declaring a property. In fact, one reason to re-declare a property is to designate a different default value.

```
__property Alignment = {default=taCenter};
```

Note: Specifying the default value does not automatically assign that value to the property on creation of the object. You must make sure that the component's constructor assigns the necessary value. A property whose value is not set by a component's constructor assumes a zero value- that is, whatever value the property assumes when its storage memory is set to 0. Thus numeric values default to 0, Boolean values to *False*, pointers to **nil**, and so on. If there is any doubt, assign a value in the constructor method.

3.2.1.4.20 Determining What to Store

You can control whether Delphi stores each of your components' properties. By default, all properties in the published part of the class declaration are stored. You can choose not to store a given property at all, or you can designate a function that determines dynamically whether to store the property.

To control whether Delphi stores a property, add the **stored** directive to the property declaration, followed by *True*, *False*, or the

name of a Boolean function.

3.2.1.4.21 Initializing After Loading

After a component reads all its property values from its stored description, it calls a virtual method named *Loaded*, which performs any required initializations. The call to *Loaded* occurs before the form and its controls are shown, so you do not need to worry about initialization causing flicker on the screen.

To initialize a component after it loads its property values, override the *Loaded* method.

Note: The first thing to do in any *Loaded* method is call the inherited *Loaded* method. This ensures that any inherited properties are correctly initialized before you initialize your own component.

The following code comes from the *TDatabase* component. After loading, the database tries to reestablish any connections that were open at the time it was stored, and specifies how to handle any exceptions that occur while connecting.

```
procedure TDatabase.Loaded;
begin
  inherited Loaded;                                  { call the inherited method first}
  try
    if FStreamedConnected then Open                        { reestablish connections }
    else CheckSessionName(False);
  except
    if csDesigning in ComponentState then                          { at design time... }
      Application.HandleException(Self)              { let Delphi handle the exception }
    else raise;                                                  { otherwise, reraise }
  end;
end;
```

3.2.1.4.22 Storing and Loading Unpublished Properties

By default, only published properties are loaded and saved with a component. However, it is possible to load and save unpublished properties. This allows you to have persistent properties that do not appear in the Object Inspector. It also allows components to store and load property values that Delphi does not know how to read or write because the value of the property is too complex. For example, the *TStrings* object can't rely on Delphi's automatic behavior to store and load the strings it represents and must use the following mechanism.

You can save unpublished properties by adding code that tells Delphi how to load and save your property's value.

To write your own code to load and save properties, use the following steps:

1. Create methods to store and load the property value (see page 1256).
2. Override the DefineProperties method (see page 1257), passing those methods to a filer object.

3.2.1.4.23 Creating Methods to Store and Load Property Values

To store and load unpublished properties, you must first create a method to store your property value and another to load your property value. You have two choices:

- Create a method of type *TWriterProc* to store your property value and a method of type *TReaderProc* to load your property value. This approach lets you take advantage of Delphi's built-in capabilities for saving and loading simple types. If your property value is built out of types that Delphi knows how to save and load, use this approach.
- Create two methods of type *TStreamProc*, one to store and one to load your property's value. *TStreamProc* takes a stream as an argument, and you can use the stream's methods to write and read your property values.

For example, consider a property that represents a component that is created at runtime. Delphi knows how to write this value, but does not do so automatically because the component is not created in the form designer. Because the streaming system can already load and save components, you can use the first approach. The following methods load and store the dynamically

created component that is the value of a property named *MyCompProperty*:

```
procedure TSampleComponent.LoadCompProperty(Reader: TReader);
begin
  if Reader.ReadBoolean then
    MyCompProperty := Reader.ReadComponent(nil);
end;
procedure TSampleComponent.StoreCompProperty(Writer: TWriter);
begin
  Writer.WriteBoolean(MyCompProperty <> nil);
  if MyCompProperty <> nil then
    Writer.WriteComponent(MyCompProperty);
end;
void __fastcall TSampleComponent::LoadCompProperty(TReader *Reader)
{
  if (Reader->ReadBoolean())
    MyCompProperty = Reader->ReadComponent(NULL);
}
void __fastcall TSampleComponent::StoreCompProperty(TWriter *Writer)
{
  if (MyCompProperty)
  {
    Writer->WriteBoolean(true);
    Writer->WriteComponent(MyCompProperty);
  }
  else
    Writer->WriteBoolean(false);
}
```

3.2.1.4.24 Overriding the DefineProperties Method

Once you have created methods to store and load your property value, you can override the component's *DefineProperties* method. Delphi calls this method when it loads or stores the component. In the *DefineProperties* method, you must call the *DefineProperty* method or the *DefineBinaryProperty* method of the current filer, passing it the method to use for loading or saving your property value. If your load and store methods are of type *TWriterProc* and type *TReaderProc*, then you call the filer's *DefineProperty* method. If you created methods of type *TStreamProc*, call *DefineBinaryProperty* instead.

No matter which method you use to define the property, you pass it the methods that store and load your property value as well as a boolean value indicating whether the property value needs to be written. If the value can be inherited or has a default value, you do not need to write it.

For example, given the *LoadCompProperty* method of type *TReaderProc* and the *StoreCompProperty* method of type *TWriterProc*, you would override *DefineProperties* as follows:

```
procedure TSampleComponent.DefineProperties(Filer: TFiler);
  function DoWrite: Boolean;
  begin
    if Filer.Ancestor <> nil then { check Ancestor for an inherited value }
    begin
      if TSampleComponent(Filer.Ancestor).MyCompProperty = nil then
        Result := MyCompProperty <> nil
      else   if (MyCompProperty = nil) or
         (TMy5Comp(Filer.Ancestor).MyCompProperty.Name <> MyCompProperty.Name) then
        Result := True
      else Result := False;
    end
    else { no inherited value -- check for default (nil) value }
      Result := MyCompProperty <> nil;
  end;
begin
  inherited; { allow base classes to define properties }
  Filer.DefineProperty('MyCompProperty', LoadCompProperty, StoreCompProperty, DoWrite);
end;
void __fastcall TSampleComponent::DefineProperties(TFiler *Filer)
```

```
{
  // before we do anything, let the base class define its properties.
  // Note that this example assumes that TSampleComponent derives directly from TComponent
  TComponent::DefineProperties(Filer);
  bool WriteValue;
  if (Filer->Ancestor) // check for inherited value
  {
    if ((TSampleComponent *)Filer->Ancestor)->MyCompProperty == NULL)
      WriteValue = (MyCompProperty != NULL);
    else if ((MyCompProperty == NULL) ||
            (((TSampleComponent *)Filer->Ancestor)->MyCompProperty->Name !=
                                                    MyCompProperty->Name))
      WriteValue = true;
    else WriteValue = false;
  }
  else // no inherited value, write property if not null
    WriteValue = (MyCompProperty != NULL);
  Filer->DefineProperty("MyCompProperty ",LoadCompProperty,StoreCompProperty, WriteValue);
end;
```

3.2.1.5 Customizing a grid

Topics

Name	Description
Accessing the Day, Month, and Year (see page 1260)	An encoded numeric date is fine for applications, but humans prefer to work with days, months, and years. You can provide alternate access to those elements of the stored, encoded date by creating properties. Because each element of the date (day, month, and year) is an integer, and because setting each requires encoding the date when set, you can avoid duplicating the code each time by sharing the implementation methods for all three properties. That is, you can write two methods, one to read an element and one to write one, and use those methods to get and set all... more (see page 1260)
Changing Initial Values (see page 1262)	A calendar is essentially a grid with a fixed number of rows and columns, although not all the rows always contain dates. For this reason, you have not published the grid properties *ColCount* and *RowCount*, because it is highly unlikely that users of the calendar will want to display anything other than seven days per week. You still must set the initial values of those properties so that the week always has seven days, however. To change the initial values of the component's properties, override the constructor to set the desired values. The constructor must be virtual. Remember that... more (see page 1262)
Creating and registering the component (Grid) (see page 1263)	You create every component the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. Creating a new component. (see page 1317)

Customizing a Grid: Overview (see page 1264)	The component library provides abstract components you can use as the basis for customized components. The most important of these are grids and list boxes. The following topics describe how to create a small one month calendar from the basic grid component, *TCustomGrid*: • Creating and registering the component (see page 1263) • Publishing inherited properties (see page 1271) • Changing initial values (see page 1262) • Resizing the cells (see page 1272) • Filling in the cells (see page 1265) • Navigating months and years (see page 1269) • Navigating days (see page 1269) In VCL applications, the resulting component is similar to the *TCalendar* component on the Samples category of the Tool palette. See Specifying the palette page (see page 1356).
Excluding Blank Cells (see page 1265)	As the calendar is written, the user can select a blank cell, but the date does not change. It makes sense, then, to disallow selection of the blank cells. To control whether a given cell is selectable, override the *SelectCell* method of the grid. *SelectCell* is a function that takes a column and row as parameters, and returns a Boolean value indicating whether the specified cell is selectable. You can override *SelectCell* to return *False* if the cell does not contain a valid date:
Filling in the Cells (see page 1265)	A grid control fills in its contents cell-by-cell. In the case of the calendar, that means calculating which date, if any, belongs in each cell. The default drawing for grid cells takes place in a virtual method called *DrawCell*. To fill in the contents of grid cells, override the *DrawCell* method. The easiest part to fill in is the heading cells in the fixed row. The runtime library contains an array with short day names, so for the calendar, use the appropriate one for each column:
Generating the Day Numbers (see page 1266)	Putting numbers into the calendar involves several considerations. The number of days in the month depends on which month it is, and whether the given year is a leap year. In addition, months start on different days of the week, dependent on the month and year. Use the *IsLeapYear* function to determine whether the year is a leap year. Use the *MonthDays* array in the SysUtils unit to get the number of days in the month. Once you have the information on leap years and days per month, you can calculate where in the grid the individual dates go.... more (see page 1266)
Moving the Selection (see page 1268)	The inherited behavior of a grid handles moving the selection in response to either arrow keys or clicks, but if you want to change the selected day, you need to modify that default behavior. To handle movements within the calendar, override the *Click* method of the grid. When you override a method such as *Click* that is tied in with user interactions, you will nearly always include a call to the inherited method, so as not to lose the standard behavior. The following is an overridden *Click* method for the calendar grid. Be sure to add the declaration of *Click... more (see page 1268)*
Navigating Days (see page 1269)	Within a given month, there are two obvious ways to navigate among the days. The first is to use the arrow keys, and the other is to respond to clicks of the mouse. The standard grid component handles both as if they were clicks. That is, an arrow movement is treated like a click on an adjacent cell. The process of navigating days consists of • Moving the selection (see page 1268) • Providing an OnChange event (see page 1270) • Excluding blank cells (see page 1265)

Navigating Months and Years (see page 1269)	Properties are useful for manipulating components, especially at design time. But sometimes there are types of manipulations that are so common or natural, often involving more than one property, that it makes sense to provide methods to handle them. One example of such a natural manipulation is a "next month" feature for a calendar. Handling the wrapping around of months and incrementing of years is simple, but very convenient for the developer using the component. The only drawback to encapsulating common manipulations into methods is that methods are only available at runtime. However, such manipulations are generally only cumbersome when... more (see page 1269)
Providing an OnChange Event (see page 1270)	Now that users of the calendar can change the date within the calendar, it makes sense to allow applications to respond to those changes.
Publishing Inherited Properties (Grid) (see page 1271)	The abstract grid component, *TCustomGrid*, provides a large number of **protected** properties. You can choose which of those properties you want to make available to users of the calendar control. To make inherited protected properties available to users of your components, redeclare the properties in the **published** part of your component's declaration. For the calendar control, publish the following properties and events, as shown here:
Resizing the Cells (see page 1272)	**Note:** When a user or application changes the size of a window or control, Windows sends a message called *WM_SIZE* to the affected window or control so it can adjust any settings needed to later paint its image in the new size. Your VCL component can respond to that message by altering the size of the cells so they all fit inside the boundaries of the control. To respond to the *WM_SIZE* message, you will add a message-handling method to the component. Creating a message-handling method is described in detail in the section Creating new message handlers. (see page 1304) In this case,... more (see page 1272)
Selecting the Current Day (see page 1274)	Now that you have numbers in the calendar cells, it makes sense to move the selection highlighting to the cell containing the current day. By default, the selection starts on the top left cell, so you need to set the *Row* and *Column* properties both when constructing the calendar initially and when the date changes. To set the selection on the current day, change the *UpdateCalendar* method to set *Row* and *Column* before calling *Refresh*:
Storing the Internal Date (see page 1274)	To store the date for the calendar, you need a private field to hold the date and a runtime-only property that provides access to that date.
Tracking the Date (see page 1275)	For the calendar control to be useful, users and applications must have a mechanism for setting the day, month, and year. Delphi stores dates and times in variables of type *TDateTime*. *TDateTime* is an encoded numeric representation of the date and time, which is useful for programmatic manipulation, but not convenient for human use. You can therefore store the date in encoded form, providing runtime access to that value, but also provide *Day*, *Month*, and *Year* properties that users of the calendar component can set at design time. Tracking the date in the calendar consists of the... more (see page 1275)

3.2.1.5.1 Accessing the Day, Month, and Year

An encoded numeric date is fine for applications, but humans prefer to work with days, months, and years. You can provide alternate access to those elements of the stored, encoded date by creating properties.

Because each element of the date (day, month, and year) is an integer, and because setting each requires encoding the date when set, you can avoid duplicating the code each time by sharing the implementation methods for all three properties. That is, you can write two methods, one to read an element and one to write one, and use those methods to get and set all three properties.

To provide design-time access to the day, month, and year, you do the following:

1. Declare the three properties, assigning each a unique **index** number:

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    property Day: Integer index 3 read GetDateElement write SetDateElement;
    property Month: Integer index 2 read GetDateElement write SetDateElement;
```

```
    property Year: Integer index 1 read GetDateElement write SetDateElement;
  .
  .
  .
class PACKAGE TSampleCalendar : public TCustomGrid
{
  .
  .
  .
public:
    __property int Day = {read=GetDateElement, write=SetDateElement, index=3,
      nodefault};
    __property int Month = {read=GetDateElement, write=SetDateElement, index=2,
nodefault};
    __property int Year = {read=GetDateElement, write=SetDateElement, index=1,
nodefault};
};
```

2. Declare and write the implementation methods, setting different elements for each index value:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    function GetDateElement(Index: Integer): Integer;         { note the Index parameter }
    procedure SetDateElement(Index: Integer; Value: Integer);
  .
  .
  .
function TSampleCalendar.GetDateElement(Index: Integer): Integer;
var
  AYear, AMonth, ADay: Word;
begin
  DecodeDate(FDate, AYear, AMonth, ADay);             { break encoded date into elements }
  case Index of
    1: Result := AYear;
    2: Result := AMonth;
    3: Result := ADay;
    else Result := -1;
  end;
end;
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
var
  AYear, AMonth, ADay: Word;
begin
  if Value > 0 then                                      { all elements must be positive }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                  { get current date elements }
    case Index of                                     { set new element depending on Index }
      1: AYear := Value;
      2: AMonth := Value;
      3: ADay := Value;
      else Exit;
    end;
    FDate := EncodeDate(AYear, AMonth, ADay);                { encode the modified date }
    Refresh;                                            { update the visible calendar }
  end;
end;
// header file
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    int __fastcall GetDateElement(int Index);           // note the Index parameter
    void __fastcall SetDateElement(int Index, int Value);
    .
    .
    .
};
```

```
// implementation file
int __fastcall TSampleCalendar::GetDateElement(int Index)
{
  unsigned short AYear, AMonth, ADay;
  int result;
  FDate.DecodeDate(&AYear, &AMonth, &ADay);            // break encoded date into elements
  switch (Index)
  {
    case 1: result = AYear;  break;
    case 2: result = AMonth; break;
    case 3: result = ADay;   break;
    default: result = -1;
  }
  return result;
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  unsigned short AYear, AMonth, ADay;
  if (Value > 0)                                        // all elements must be positive
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);     // get current date elements
    switch (Index)
    {
      case 1: AYear = Value;     break;
      case 2: AMonth = Value;    break;
      case 3: ADay = Value;      break;
      default: return;
    }
  }
  FDate = TDateTime(AYear, AMonth, ADay);         // encode the modified date
  Refresh();                                      // update the visible calendar
}
```

Now you can set the calendar's day, month, and year at design time using the Object Inspector or at runtime using code. Of course, you have not yet added the code to paint the dates into the cells, but now you have the needed data.

3.2.1.5.2 Changing Initial Values

A calendar is essentially a grid with a fixed number of rows and columns, although not all the rows always contain dates. For this reason, you have not published the grid properties *ColCount* and *RowCount*, because it is highly unlikely that users of the calendar will want to display anything other than seven days per week. You still must set the initial values of those properties so that the week always has seven days, however.

To change the initial values of the component's properties, override the constructor to set the desired values. The constructor must be virtual.

Remember that you need to add the constructor to the **public** part of the component's object declaration, then write the new constructor in the **implementation** part of the component's unit. The first statement in the new constructor should always be a call to the inherited constructor. Then add the StdCtrls unit to the **uses** clause.

```
type
  TSampleCalendar = class(TCustomGrid)
  public
    constructor Create(AOwner: TComponent); override;
  .
  .
  .
  end;
.
.
.
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
```

```
  inherited Create(AOwner);                                    { call inherited constructor }
  ColCount := 7;                                                  { always seven days/week }
  RowCount := 7;                                          { always six weeks plus the headings }
  FixedCols := 0;                                                          { no row labels }
  FixedRows := 1;                                                { one row for day names }
  ScrollBars := ssNone;                                             { no need to scroll }
  Options := Options - [goRangeSelect] + [goDrawFocusSelected];  {disable range selection}
end;
//header file
class PACKAGE TSampleCalendar : public TCustomGrid
{
protected:
    virtual void __fastcall DrawCell(int ACol, int ARow, const Windows::TRect &Rect,
      TGridDrawState AState);
    .
    .
    .
public:
    __fastcall TSampleCalendar(TComponent *Owner);   // the added constructor
    .
    .
    .
};
//implementation file
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) : TCustomGrid(Owner)
{
  ColCount = 7;
  RowCount = 7;
  FixedCols = 0;
  FixedRows = 1;
  ScrollBars = ssNone;
  Options = (Options >> goRangeSelect) << goDrawFocusSelected;
}
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const Windows::TRect
  &ARect, TGridDrawState AState)
{
}
```

The calendar now has seven columns and seven rows, with the top row fixed, or nonscrolling.

3.2.1.5.3 Creating and registering the component (Grid)

You create every component the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. Creating a new component. (see page 1317)

For this example, follow the general procedure for creating a component, with these specifics:

1. Save the component's unit as *CalSamp*.
2. Derive a new component type called *TSampleCalendar*, descended from *TCustomGrid*.
3. Register *TSampleCalendar* on the Samples category of the Tool palette.

The resulting unit descending from *TCustomGrid* in a VCL application should look like this:

```
unit CalSamp;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;
type
  TSampleCalendar = class(TCustomGrid)
  end;
procedure Register;
implementation
procedure Register;
begin
```

```
  RegisterComponents('Samples', [TSampleCalendar]);
end;
end.
#include <vcl\vcl.h>
#pragma hdrstop
#include "CalSamp.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline TSampleCalendar *ValidCtrCheck()
{
  return new TSampleCalendar(NULL);
}
//---------------------------------------------------------------------------
namespace Calsamp
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TSampleCalendar)};
    RegisterComponents("Samples", classes, 0); //Use a different page in CLX applications
  }
}
#ifndef CalSampH
#define CalSampH
//---------------------------------------------------------------------------
#include <vcl\sysutils.hpp>
#include <vcl\controls.hpp>
#include <vcl\classes.hpp>
#include <vcl\forms.hpp>
#include <vcl\grids.hpp>
//---------------------------------------------------------------------------
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

If you install the calendar component now, you will find that it appears on the Samples category. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

Note: While you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the *DrawCell* method is described in Filling in the cells (see page 1265)

3.2.1.5.4 Customizing a Grid: Overview

The component library provides abstract components you can use as the basis for customized components. The most important of these are grids and list boxes. The following topics describe how to create a small one month calendar from the basic grid component, *TCustomGrid*:

- Creating and registering the component (see page 1263)
- Publishing inherited properties (see page 1271)
- Changing initial values (see page 1262)
- Resizing the cells (see page 1272)
- Filling in the cells (see page 1265)

- Navigating months and years (see page 1269)
- Navigating days (see page 1269)

In VCL applications, the resulting component is similar to the *TCalendar* component on the Samples category of the Tool palette. See Specifying the palette page (see page 1356).

3.2.1.5.5 Excluding Blank Cells

As the calendar is written, the user can select a blank cell, but the date does not change. It makes sense, then, to disallow selection of the blank cells.

To control whether a given cell is selectable, override the *SelectCell* method of the grid.

SelectCell is a function that takes a column and row as parameters, and returns a Boolean value indicating whether the specified cell is selectable.

You can override *SelectCell* to return *False* if the cell does not contain a valid date:

```
function TSampleCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if DayNum(ACol, ARow) = -1 then Result := False            { -1 indicates invalid date }
  else Result := inherited SelectCell(ACol, ARow);      { otherwise, use inherited value }
end;
bool __fastcall TSampleCalendar::SelectCell(int ACol, int ARow)
{
  if (DayNum(ACol,ARow) == -1) return false;          // -1 indicates invalid date
  else return TCustomGrid::SelectCell(ACol, ARow);    // otherwise, use inherited value
}
```

Now if the user clicks a blank cell or tries to move to one with an arrow key, the calendar leaves the current cell selected.

3.2.1.5.6 Filling in the Cells

A grid control fills in its contents cell-by-cell. In the case of the calendar, that means calculating which date, if any, belongs in each cell. The default drawing for grid cells takes place in a virtual method called *DrawCell*.

To fill in the contents of grid cells, override the *DrawCell* method.

The easiest part to fill in is the heading cells in the fixed row. The runtime library contains an array with short day names, so for the calendar, use the appropriate one for each column:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
      override;
  end;
.
.
.
procedure TSampleCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect;
  AState: TGridDrawState);
begin
  if ARow = 0 then
    Canvas.TextOut(ARect.Left, ARect.Top, ShortDayNames[ACol + 1]);    { use RTL strings }
end;
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const Windows::TRect &ARect,
  TGridDrawState AState)
{
  String TheText;
  int TempDay;
  if (ARow == 0) TheText = ShortDayNames[ACol + 1];
```

```
  else
  {
    TheText = "";
    TempDay = DayNum(ACol, ARow);                    // DayNum is defined later
    if (TempDay != -1) TheText = IntToStr(TempDay);
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left
   - Canvas->TextWidth(TheText)) / 2,
  ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
}
```

3.2.1.5.7 Generating the Day Numbers

Putting numbers into the calendar involves several considerations. The number of days in the month depends on which month it is, and whether the given year is a leap year. In addition, months start on different days of the week, dependent on the month and year. Use the *IsLeapYear* function to determine whether the year is a leap year. Use the *MonthDays* array in the SysUtils unit to get the number of days in the month.

Once you have the information on leap years and days per month, you can calculate where in the grid the individual dates go. The calculation is based on the day of the week the month starts on.

Because you will need the month-offset number for each cell you fill in, the best practice is to calculate it once when you change the month or year, then refer to it each time. You can store the value in a class field, then update that field each time the date changes.

To fill in the days in the proper cells, you do the following:

1. Add a month-offset field to the object and a method that updates the field value:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FMonthOffset: Integer;                                  { storage for the offset }
  .
  .
  .
  protected
    procedure UpdateCalendar; virtual;                    { property for offset access }
  end;
.
.
.
procedure TSampleCalendar.UpdateCalendar;
var
  AYear, AMonth, ADay: Word;
  FirstDate: TDateTime;                              { date of the first day of the month }
begin
  if FDate <> 0 then                               { only calculate offset if date is valid }
  begin
    DecodeDate(FDate, AYear, AMonth, ADay);                      { get elements of date }
    FirstDate := EncodeDate(AYear, AMonth, 1);                     { date of the first }
    FMonthOffset := 2 - DayOfWeek(FirstDate);        { generate the offset into the grid }
  end;
  Refresh;                                                  { always repaint the control }
end;
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    int FMonthOffset;                           // storage for the offset
    .
    .
    .
protected:
    virtual void __fastcall UpdateCalendar(void);
```

```
    .
    .
    .
};
void __fastcall TSampleCalendar::UpdateCalendar(void)
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;                          // date of first day of the month
  if ((int)FDate != 0)                          // only calculate offset if date is valid
  {
    FDate.DecodeDate(&AYear, &AMonth, &ADay);  // get elements of date
    FirstDate = TDateTime(AYear, AMonth, 1);   // date of the first
    FMonthOffset = 2 - FirstDate.DayOfWeek();  // generate the offset into the grid
  }
  Refresh();                                    // always repaint the control
}
```

2. Add statements to the constructor and the *SetCalendarDate* and *SetDateElement* methods that call the new update method whenever the date changes:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                                    { this is already here }
  .                                                          { other initializations here }
  .
  .
  UpdateCalendar;                                                { set proper offset }
end;
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin  FDate := Value;                                                { this was already here }
  UpdateCalendar;                                       { this previously called Refresh }
end;
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
  .
  .
  .
    FDate := EncodeDate(AYear, AMonth, ADay);              { encode the modified date }
    UpdateCalendar;                                     { this previously called Refresh }
  end;
end;
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner)
  : TCustomGrid(Owner)
{
  .
  .
  .
  UpdateCalendar();
}
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                                  // this was already here
  UpdateCalendar();                               // this previously called Refresh
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  .
  .
  .
  FDate = TDateTime(AYear, AMonth, ADay);        // this was already here
  UpdateCalendar();                              // this previously called Refresh
}
```

3. Add a method to the calendar that returns the day number when passed the row and column coordinates of a cell:

```
function TSampleCalendar.DayNum(ACol, ARow: Integer): Integer;
begin
  Result := FMonthOffset + ACol + (ARow - 1) * 7;          { calculate day for this cell }
```

```
  if (Result < 1) or (Result > MonthDays[IsLeapYear(Year), Month]) then
    Result := -1;                                              { return -1 if invalid }
end;
int __fastcall TSampleCalendar::DayNum(int ACol, int ARow)
{
  int result = FMonthOffset + ACol + (ARow - 1) * 7;       // calculate day for this cell
  if ((result < 1)||(result > MonthDays[IsLeapYear(Year)][Month]))
    result = -1;   // return -1 if invalid
  return result;
}
```

Remember to add the declaration of *DayNum* to the component's type declaration.

4. Now that you can calculate where the dates go, you can update *DrawCell* to fill in the dates:

```
procedure TCalendar.DrawCell(ACol, ARow: Longint; ARect: TRect; AState: TGridDrawState);
var
  TheText: string;
  TempDay: Integer;
begin
  if ARow = 0 then                                        { if this is the header row ...}
    TheText := ShortDayNames[ACol + 1]                           { just use the day name }
  else begin
    TheText := '';                                          { blank cell is the default }
    TempDay := DayNum(ACol, ARow);                           { get number for this cell }
    if TempDay <> -1 then TheText := IntToStr(TempDay);       { use the number if valid }
  end;
  with ARect, Canvas do
    TextRect(ARect, Left + (Right - Left - TextWidth(TheText)) div 2,
      Top + (Bottom - Top - TextHeight(TheText)) div 2, TheText);
end;
void __fastcall TSampleCalendar::DrawCell(int ACol, int ARow, const TRect &ARect,
  TGridDrawState AState)
{
  String TheText;
  int TempDay;
  if (ARow == 0)                                    // this is the header row
    TheText = ShortDayNames[ACol + 1];              // just use the day name
  else
  {
    TheText = "";                                   // blank cell is the default
    TempDay = DayNum(ACol, ARow);                   // get number for this cell
    if (TempDay != -1) TheText = IntToStr(TempDay); // use the number if valid
  }
  Canvas->TextRect(ARect, ARect.Left + (ARect.Right - ARect.Left -
    Canvas->TextWidth(TheText)) / 2,
  ARect.Top + (ARect.Bottom - ARect.Top - Canvas->TextHeight(TheText)) / 2, TheText);
}
```

Now if you reinstall the calendar component and place one on a form, you will see the proper information for the current month.

3.2.1.5.8 Moving the Selection

The inherited behavior of a grid handles moving the selection in response to either arrow keys or clicks, but if you want to change the selected day, you need to modify that default behavior.

To handle movements within the calendar, override the *Click* method of the grid.

When you override a method such as *Click* that is tied in with user interactions, you will nearly always include a call to the inherited method, so as not to lose the standard behavior.

The following is an overridden *Click* method for the calendar grid. Be sure to add the declaration of *Click* to *TSampleCalendar*, including the **override** directive afterward.

```
procedure TSampleCalendar.Click;
```

```
var
  TempDay: Integer;
begin
  inherited Click;                          { remember to call the inherited method! }
  TempDay := DayNum(Col, Row);              { get the day number for the clicked cell }
  if TempDay <> -1 then Day := TempDay;                        { change day if valid }
end;
void __fastcall TSampleCalendar::Click()
{
  int TempDay = DayNum(Col, Row);           // get the day number for the clicked cell
  if (TempDay != -1) Day = TempDay;         // change day if valid
}
```

3.2.1.5.9 Navigating Days

Within a given month, there are two obvious ways to navigate among the days. The first is to use the arrow keys, and the other is to respond to clicks of the mouse. The standard grid component handles both as if they were clicks. That is, an arrow movement is treated like a click on an adjacent cell.

The process of navigating days consists of

- Moving the selection (see page 1268)
- Providing an OnChange event (see page 1270)
- Excluding blank cells (see page 1265)

3.2.1.5.10 Navigating Months and Years

Properties are useful for manipulating components, especially at design time. But sometimes there are types of manipulations that are so common or natural, often involving more than one property, that it makes sense to provide methods to handle them. One example of such a natural manipulation is a "next month" feature for a calendar. Handling the wrapping around of months and incrementing of years is simple, but very convenient for the developer using the component.

The only drawback to encapsulating common manipulations into methods is that methods are only available at runtime. However, such manipulations are generally only cumbersome when performed repeatedly, and that is fairly rare at design time.

For the calendar, add the following four methods for next and previous month and year. Each of these methods uses the *IncMonth* function in a slightly different manner to increment or decrement *CalendarDate*, by increments of a month or a year.

```
procedure TCalendar.NextMonth;
begin
  CalendarDate := IncMonth(CalendarDate, 1);
end;
procedure TCalendar.PrevMonth;
begin
  CalendarDate := IncMonth(CalendarDate, -1);
end;
procedure TCalendar.NextYear;
begin
  CalendarDate := IncMonth(CalendarDate, 12);
end;
procedure TCalendar.PrevYear;
begin
  CalendarDate := DecodeDate(IncMonth(CalendarDate, -12);
end;
void __fastcall TSampleCalendar::NextMonth()
{
  CalendarDate = IncMonth(CalendarDate, 1);
}
void __fastcall TSampleCalendar::PrevMonth()
{
  CalendarDate = IncMonth(CalendarDate, -1);
```

```
}
void __fastcall TSampleCalendar::NextYear()
{
  CalendarDate = IncMonth(CalendarDate, 12);
}
void __fastcall TSampleCalendar::PrevYear()
{
  CalendarDate = IncMonth(CalendarDate, -12);
}
```

Be sure to add the declarations of the new methods to the class declaration.

Now when you create an application that uses the calendar component, you can easily implement browsing through months or years.

3.2.1.5.11 Providing an OnChange Event

Now that users of the calendar can change the date within the calendar, it makes sense to allow applications to respond to those changes.

Add an OnChange event to TSampleCalendar.

1. Declare the event, a field to store the event, and a dynamic method to call the event:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  .
  .
  .
  published
    property OnChange: TNotifyEvent read FOnChange write FOnChange;
  .
  .
  .
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    TNotifyEvent FOnChange;
    .
    .
    .
protected:
    virtual void __fastcall Change();
__published:
    __property TNotifyEvent OnChange = {read=FOnChange, write=FOnChange};
    .
    .
    .
}
```

2. Write the *Change* method:

```
procedure TSampleCalendar.Change;
begin
  if Assigned(FOnChange) then FOnChange(Self);
end;
void __fastcall TSampleCalendar::Change()
{
  if(FOnChange != NULL) FOnChange(this);
}
```

3. Add statements calling *Change* to the end of the *SetCalendarDate* and *SetDateElement* methods:

```
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;
  UpdateCalendar;
  Change;                                                { this is the only new statement }
end;
procedure TSampleCalendar.SetDateElement(Index: Integer; Value: Integer);
begin
    .                                                  { many statements setting element values }
    .
    .
    FDate := EncodeDate(AYear, AMonth, ADay);
    UpdateCalendar;
    Change;                                                                  { this is new }
  end;
end;
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;
  UpdateCalendar();
  Change();              // this is the only new statement
}
void __fastcall TSampleCalendar::SetDateElement(int Index, int Value)
{
  .
  .
  .    // many statements setting element values
  FDate = TDateTime(AYear, AMonth, ADay);
  UpdateCalendar();
  Change();              // this is new
}
```

Applications using the calendar component can now respond to changes in the date of the component by attaching handlers to the *OnChange* event.

3.2.1.5.12 Publishing Inherited Properties (Grid)

The abstract grid component, *TCustomGrid*, provides a large number of **protected** properties. You can choose which of those properties you want to make available to users of the calendar control.

To make inherited protected properties available to users of your components, redeclare the properties in the **published** part of your component's declaration.

For the calendar control, publish the following properties and events, as shown here:

```
type
  TSampleCalendar = class(TCustomGrid)
  published
    property Align;  { publish properties }
    property BorderStyle;
    property Color;
    property Font;
    property GridLineWidth;
    property ParentColor;
    property ParentFont;
    property OnClick;  { publish events }
    property OnDblClick;
    property OnDragDrop;
    property OnDragOver;
    property OnEndDrag;
    property OnKeyDown;
    property OnKeyPress;
```

```
    property OnKeyUp;
  end;
class PACKAGE TSampleCalendar : public TCustomGrid
{
.
.
.
__published:
    __property Align ;                    // publish properties
    __property BorderStyle ;
    __property Color ;
    __property Font ;
    __property GridLineWidth ;
    __property ParentColor ;
    __property ParentFont ;
    __property OnClick ;                   // publish events
    __property OnDblClick ;
    __property OnDragDrop ;
    __property OnDragOver ;
    __property OnEndDrag ;
    __property OnKeyDown ;
    __property OnKeyPress ;
    __property OnKeyUp ;
};
```

There are a number of other properties you could also publish, but which do not apply to a calendar, such as the *Options* property that would enable the user to choose which grid lines to draw.

If you install the modified calendar component to the Tool palette and use it in an application, you will find many more properties and events available in the calendar, all fully functional. You can now start adding new capabilities of your own design.

3.2.1.5.13 Resizing the Cells

Note: When a user or application changes the size of a window or control, Windows sends a message called *WM_SIZE* to the affected window or control so it can adjust any settings needed to later paint its image in the new size. Your VCL component can respond to that message by altering the size of the cells so they all fit inside the boundaries of the control. To respond to the *WM_SIZE* message, you will add a message-handling method to the component.

Creating a message-handling method is described in detail in the section Creating new message handlers. (see page 1304)

In this case, the calendar control needs a response to *WM_SIZE*, so add a protected method called *WMSize* to the control indexed to the *WM_SIZE* message, then write the method so that it calculates the proper cell size to allow all cells to be visible in the new size:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
  .
  .
  .
  end;
.
.
.
procedure TSampleCalendar.WMSize(var Message: TWMSize);
var
  GridLines: Integer;                                    { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;                { calculate combined size of all lines }
  DefaultColWidth := (Message.Width - GridLines) div 7;     { set new default cell width }
  DefaultRowHeight := (Message.Height - GridLines) div 7;             { and cell height }
end;
```

```
//header file
class PACKAGE TSampleCalendar : public TCustomGrid
{
.
.
.
protected:
    void __fastcall WMSize(TWMSize &Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_SIZE, TWMSize, WMSize)
END_MESSAGE_MAP(TCustomGrid)
};
//implementation file
void __fastcall TSampleCalendar::WMSize(TWMSize &Message)
{
  int GridLines;                                  // temporary local variable
  GridLines = 6 * GridLineWidth;                  // calculated combined size of all lines
  DefaultColWidth = (Message.Width - GridLines) / 7;    // set new default cell width
  DefaultRowHeight = (Message.Height - GridLines) / 7;  // and cell height
}
```

Now when the calendar is resized, it displays all the cells in the largest size that will fit in the control.

In this case, the calendar control needs to override *BoundsChanged* so that it calculates the proper cell size to allow all cells to be visible in the new size:

```
type
  TSampleCalendar = class(TCustomGrid)
  protected
    procedure BoundsChanged; override;
  .
  .
  .
  end;
.
.
.
procedure TSampleCalendar.BoundsChanged;
var
  GridLines: Integer;                                    { temporary local variable }
begin
  GridLines := 6 * GridLineWidth;                 { calculate combined size of all lines }
  DefaultColWidth := (Width - GridLines) div 7;     { set new default cell width }
  DefaultRowHeight := (Height - GridLines) div 7;             { and cell height }
  inherited; {now call the inherited method }
end;
//header file
class PACKAGE TSampleCalendar : public TCustomGrid
{
.
.
.
protected:
    void __fastcall BoundsChanged(void);
};
//implementation file
void __fastcall TSampleCalendar::BoundsChanged(void)
{
  int GridLines;                                  // temporary local variable
  GridLines = 6 * GridLineWidth;                  // calculated combined size of all lines
  DefaultColWidth = (Width - GridLines) / 7;    // set new default cell width
  DefaultRowHeight = (Height - GridLines) / 7;  // and cell height
  TCustomGrid::BoundsChanged(); // now call the inherited method
}
```

3.2.1.5.14 Selecting the Current Day

Now that you have numbers in the calendar cells, it makes sense to move the selection highlighting to the cell containing the current day. By default, the selection starts on the top left cell, so you need to set the *Row* and *Column* properties both when constructing the calendar initially and when the date changes.

To set the selection on the current day, change the *UpdateCalendar* method to set *Row* and *Column* before calling *Refresh*:

```
procedure TSampleCalendar.UpdateCalendar;
begin
  if FDate <> 0 then
  begin
    .  { existing statements to set FMonthOffset }
    .
    .
    Row := (ADay - FMonthOffset) div 7 + 1;
    Col := (ADay - FMonthOffset) mod 7;
  end;
  Refresh; { this is already here }
end;
void __fastcall TSampleCalendar::UpdateCalendar(void)
{
  unsigned short AYear, AMonth, ADay;
  TDateTime FirstDate;
  if ((int) FDate != 0)
  {
  .
  .
  .                                                  // existing statements to set FMonthOffset
    Row = (ADay - FMonthOffset) / 7 + 1;
    Col = (ADay - FMonthOffset) % 7;
  }
  Refresh();                                        // this is already here
}
```

Note that you are now reusing the *ADay* variable previously set by decoding the date.

3.2.1.5.15 Storing the Internal Date

To store the date for the calendar, you need a private field to hold the date and a runtime-only property that provides access to that date.

Adding the internal date to the calendar requires three steps:

1. Declare a private field to hold the date:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    FDate: TDateTime;
  .
  .
  .
class PACKAGE TSampleCalendar : public TCustomGrid
{
public:
    __property TDateTime CalendarDate = {read=FDate, write=SetCalendarDate, nodefault};
    .
    .
    .
};
class PACKAGE TSampleCalendar : public TCustomGrid
```

```
{
private:
    TDateTime FDate;
    .
    .
    .
};
```

2. Initialize the date field in the constructor:

```
constructor TSampleCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);         { this is already here }
  .                                  { other initializations here }
  .
  .
  FDate := Date;                    { get current date from RTL }
end;
__fastcall TSampleCalendar::TSampleCalendar(TComponent *Owner) : TCustomGrid(Owner)
{
  .
  .
  .
  FDate = FDate.CurrentDate();
}
```

3. Declare a runtime property to allow access to the encoded date. You'll need a method for setting the date, because setting the date requires updating the onscreen image of the control:

```
type
  TSampleCalendar = class(TCustomGrid)
  private
    procedure SetCalendarDate(Value: TDateTime);
  public
    property CalendarDate: TDateTime read FDate write SetCalendarDate;
  .
  .
  .
procedure TSampleCalendar.SetCalendarDate(Value: TDateTime);
begin
  FDate := Value;                { set new date value }
  Refresh;                       { update the onscreen image }
end;
class PACKAGE TSampleCalendar : public TCustomGrid
{
private:
    void __fastcall SetCalendarDate(TDateTime Value);
    .
    .
    .
};
void __fastcall TSampleCalendar::SetCalendarDate(TDateTime Value)
{
  FDate = Value;                      // Set the new date value
  Refresh();                          // Update the onscreen image
}
```

3.2.1.5.16 Tracking the Date

For the calendar control to be useful, users and applications must have a mechanism for setting the day, month, and year. Delphi stores dates and times in variables of type *TDateTime*. *TDateTime* is an encoded numeric representation of the date and time, which is useful for programmatic manipulation, but not convenient for human use.

You can therefore store the date in encoded form, providing runtime access to that value, but also provide *Day*, *Month*, and *Year* properties that users of the calendar component can set at design time.

Tracking the date in the calendar consists of the processes:

- Storing the internal date (see page 1274)
- Accessing the day (see page 1260)
- Generating the day numbers (see page 1266)
- Selecting the current day (see page 1274)

3.2.1.6 Extending the IDE

Topics

Name	Description
Adding an Action to the Action List (see page 1278)	The image index obtained in Adding an image to the image list (see page 1279) is used to create an action, as shown below. The wizard uses the *OnExecute* and *OnUpdate* events. A common scenario is for a wizard to use the *OnUpdate* event to enable or disable the action. Be sure the *OnUpdate* event returns quickly, or the user will notice that the IDE becomes sluggish after loading your wizard. The action's *OnExecute* event is similar to the wizard's *Execute* method. If you are using a menu item to invoke a form or project wizard, you might even want to have *OnExecute... more (see page 1278)*
Adding an Image to the Image List (see page 1279)	Suppose you want to add a menu item to invoke your wizard. You also want to enable the user to add a toolbar button that invokes the wizard. The first step is to add an image to the IDE's image list. The index of your image can then be used for the action, which in turn is used by the menu item and toolbar button. Create a resource file that contains a 16 by 16 bitmap resource. Add the following code to your wizard's constructor:
Creating Forms and Projects (see page 1279)	Delphi comes with a number of form and project wizards already installed, and you can write your own. The Object Repository lets you create static templates that can be used in a project, but a wizard offers much more power because it is dynamic. The wizard can prompt the user and create different kinds of files depending on the user's responses. A form or project wizard typically creates one or more new files. Instead of real files, however, it is best to create unnamed, unsaved modules. When the user saves them, the IDE prompts the user for a file name.... more (see page 1279)
Debugging a Wizard (see page 1284)	The Tools API provides you with a lot of flexibility in how your wizard interacts with the IDE. With the flexibility comes responsibility, however. It is easy to wind up with dangling pointers or other access violations. When writing wizards that use the native tools API, you can write code that causes the IDE to crash. It is also possible that you write a wizard that installs but does not act the way you want it to. One of the challenges of working with design-time code is debugging. It's an easy problem to solve, however. Because the wizard is installed... more (see page 1284)
Deleting Toolbar Buttons (see page 1284)	There is no convenient function for removing a button from a toolbar; you must send the CM_CONTROLCHANGE message, where the first parameter is the control to change, and the second parameter is zero to remove it or non-zero to add it to the toolbar. After removing the toolbar buttons, the destructor deletes the action and menu item. Deleting these items automatically removes them from the IDE's ActionList and MainMenu.
Extending the IDE (see page 1285)	You can extend and customize the IDE with your own menu items, tool bar buttons, dynamic form-creation wizards, and more, using the Open Tools API (often shortened to just Tools API). The Tools API is a suite of over 100 interfaces that interact with and control the IDE, including the main menu, the tool bars, the main action list and image list, the source editor's internal buffers, keyboard macros and bindings, forms and their components in the form editor, the debugger and the process being debugged, code completion, the message view, and the To-Do list. Using the Tools API is... more (see page 1285)
Implementing the Wizard Interfaces (see page 1286)	Every wizard class must implement at least *IOTAWizard*, which requires implementing its ancestors, too: *IOTANotifier* and *IInterface*. Form and project wizards must implement all their ancestor interfaces, namely, *IOTARepositoryWizard*, *IOTAWizard*, *IOTANotifier*, and *IInterface*. For C++, to use NotifierObject as a base class you must use multiple inheritance. Your wizard class must inherit from NotifierObject and from the wizard interfaces that you need to implement, such as IOTAWizard. Because IOTAWizard inherits from IOTANotifier and IInterface, there is an ambiguity in the derived class: functions such as AddRef() are declared in every branch of the ancestral... more (see page 1286)

Installing the Wizard Package (see page 1288)	As with any other design-time package, a wizard package must have a *Register* function. (See Registering components (see page 1352) for details about the *Register* function.) In the *Register* function, you can register any number of wizards by calling *RegisterPackageWizard*, and passing a wizard object as the sole argument, as shown below:
Interface Version Numbers (see page 1288)	If you look closely at the declarations of some of the interfaces, such as *IOTAMessageServices*, you will see that they inherit from other interfaces with similar names, such as *IOTAMessageServices50*, which inherits from *IOTAMessageServices40*. This use of version numbers helps insulate your code from changes between releases of Delphi. The Tools API follows the basic principle of COM, namely, that an interface and its GUID never change. If a new release adds features to an interface, the Tools API declares a new interface that inherits from the old one. The GUID remains the same, attached to the... more (see page 1288)
Notifying a Wizard of IDE Events (see page 1289)	An important aspect of writing a well-behaved wizard is to have the wizard respond to IDE events. In particular, any wizard that keeps track of module interfaces must know when the user closes the module, so the wizard can release the interface. To do this, the wizard needs a notifier, which means you must write a notifier class. All notifier classes implement one or more notifier interfaces. The notifier interfaces define callback methods; the wizard registers a notifier object with the Tools API, and the IDE calls back to the notifier when something important happens. Every notifier interface inherits from... more (see page 1289)
Obtaining Tools API Services (see page 1293)	To do anything useful, a wizard needs access to the IDE: its editors, windows, menus, and so on. This is the role of the service interfaces. The Tools API includes many services, such as action services to perform file actions, editor services to access the source code editor, debugger services to access the debugger, and so on. The following table summarizes all the service interfaces. ***Tools API service interfaces***
Overview of the Tools API (see page 1294)	All of the Tools API declarations reside in a single unit, ToolsAPI. To use the Tools API, you typically use the designide package, which means you must build your Tools API add-in as a design-time package or as a DLL that uses runtime packages. For information about package and library issues, see Installing the wizard package (see page 1288). The main interface for writing a Tools API extension is *IOTAWizard*, so most IDE add-ins are called wizards. C++Builder and Delphi wizards are, for the most part, interoperable. You can write and compile a wizard in Delphi, then use it in C++Builder,... more (see page 1294)
Using Editor Interfaces (see page 1295)	Every module has at least one editor interface. Some modules have several editors, such as a source (.pas) file and form description (.dfm) file. All editors implement the *IOTAEditor* interface; cast the editor to a specific type to learn what kind of editor it is. For example, to obtain the form editor interface for a unit, you can do the following:
Using Module Interfaces (see page 1296)	To obtain a module interface, start with the module services (*IOTAModuleServices*). You can query the module services for all open modules, look up a module from a file name or form name, or open a file to obtain its module interface. There are different kinds of modules for different kinds of files, such as projects, resources, and type libraries. Cast a module interface to a specific kind of module interface to learn whether the module is of that type. For example, one way to obtain the current project group interface is as follows:
Using Native IDE Objects (see page 1297)	Wizards have full access to the main menu, tool bars, action list, and image list of the IDE. (Note that the IDE's many context menus are not accessible through the Tools API.) The starting point for working with native IDE objects is the *INTAServices* interface. Use this interface to add an image to the image list, an action to the action list, a menu item to the main menu, and a button to a tool bar. You can tie the action to the menu item and tool button. When the wizard is destroyed, it must clean up the objects it... more (see page 1297)
Working with Files and Editors (see page 1297)	It is important to understand how the Tools API works with files. The main interface is *IOTAModule*. A module represents a set of logically related open files. For example, a single module represents a single unit. The module, in turn, has one or more editors, where each editor represents one file, such as the unit source (.pas) or form (.dfm or .xfm) file. The editor interfaces reflect the internal state of the IDE's editors, so a wizard can see the modified code and forms that the user sees, even if the user has not saved any changes. The following... more (see page 1297)

Writing a Wizard Class (see page 1297)	There are four kinds of wizards, where the wizard kind depends on the interfaces that the wizard class implements. The following table describes the four kinds of wizards. ***The four kinds of wizards***

3.2.1.6.1 Adding an Action to the Action List

The image index obtained in Adding an image to the image list (see page 1279) is used to create an action, as shown below. The wizard uses the *OnExecute* and *OnUpdate* events. A common scenario is for a wizard to use the *OnUpdate* event to enable or disable the action. Be sure the *OnUpdate* event returns quickly, or the user will notice that the IDE becomes sluggish after loading your wizard. The action's *OnExecute* event is similar to the wizard's *Execute* method. If you are using a menu item to invoke a form or project wizard, you might even want to have *OnExecute* call *Execute* directly.

```
NewAction := TAction.Create(nil);
NewAction.ActionList := Services.ActionList;
NewAction.Caption    := GetMenuText();
NewAction.Hint       := 'Display a silly dialog box';
NewAction.ImageIndex := ImageIndex;
NewAction.OnUpdate   := action_update;
NewAction.OnExecute  := action_execute;
action = new TAction(0);
action->ActionList = services->ActionList;
action->Caption    = GetMenuText();
action->Hint       = "Display a silly dialog box";
action->ImageIndex = image;
action->OnUpdate   = action_update;
action->OnExecute  = action_execute;
```

The menu item sets its *Action* property to the newly created action. The tricky part of creating the menu item is knowing where to insert it. The example below looks for theView menu, and inserts the new menu item as the first item in the View menu. (In general, relying on absolute position is not a good idea: you never know when another wizard might insert itself in the menu. Future versions of Delphi are likely to reorder the menu, too. A better approach is to search the menu for a menu item with a specific name. The simplistic approach follows for the sake of clarity.)

```
for I := 0 to Services.MainMenu.Items.Count - 1 do
begin
  with Services.MainMenu.Items[I] do
  begin
    if CompareText(Name, 'ViewsMenu') = 0 then
    begin
      NewItem := TMenuItem.Create(nil);
      NewItem.Action := NewAction;
      Insert(0, NewItem);
    end;
  end;
end;
for (int i = 0; i < services->MainMenu->Items->Count; ++i)
{
TMenuItem* item = services->MainMenu->Items->Items[i];
if (CompareText(item->Name, "ViewsMenu") == 0)
{
menu_item = new TMenuItem(0);
menu_item->Action = action;
item->Insert(0, menu_item);
}
}
```

By adding the action to the IDE's action list, the user can see the action when customizing the toolbars. The user can select the action and add it as a button to any toolbar. This causes a problem when your wizard is unloaded: all the tool buttons end up with dangling pointers to the non-existent action and *OnClick* event handler. To prevent access violations, your wizard must find all tool buttons that refer to its action, and remove those buttons.

See Also

Adding an Image to the Image List (see page 1279)

Deleting Toolbar Buttons (see page 1284)

3.2.1.6.2 Adding an Image to the Image List

Suppose you want to add a menu item to invoke your wizard. You also want to enable the user to add a toolbar button that invokes the wizard. The first step is to add an image to the IDE's image list. The index of your image can then be used for the action, which in turn is used by the menu item and toolbar button. Create a resource file that contains a 16 by 16 bitmap resource. Add the following code to your wizard's constructor:

```
constructor MyWizard.Create;
var
  Services: INTAServices;
  Bmp: TBitmap;
  ImageIndex: Integer;
begin
  inherited;
  Supports(BorlandIDEServices, INTAServices, Services);
  { Add an image to the image list. }
  Bmp := TBitmap.Create;
  Bmp.LoadFromResourceName(HInstance, 'Bitmap1');
  ImageIndex := Services.AddMasked(Bmp, Bmp.TransparentColor,
                                   'Tempest Software.intro wizard image');
  Bmp.Free;
end;
_di_INTAServices services;
BorlandIDEServices->Supports(services);
// Add an image to the image list.
Graphics::TBitmap* bitmap(new Graphics::TBitmap());
bitmap->LoadFromResourceName(reinterpret_cast<unsigned>(HInstance), "Bitmap1");
int image = services->AddMasked(bitmap, bitmap->TransparentColor,
                                 "Tempest Software.intro wizard image");
delete bitmap;
```

Be sure to load the resource by the name or ID you specify in the resource file. You must choose a color that will be interpreted as the background color for the image. If you don't want a background color, choose a color that does not exist in the bitmap.

See Also

Adding an Action to the Action List (see page 1278)

Deleting Toolbar Buttons (see page 1284)

3.2.1.6.3 Creating Forms and Projects

Delphi comes with a number of form and project wizards already installed, and you can write your own. The Object Repository lets you create static templates that can be used in a project, but a wizard offers much more power because it is dynamic. The wizard can prompt the user and create different kinds of files depending on the user's responses.

A form or project wizard typically creates one or more new files. Instead of real files, however, it is best to create unnamed, unsaved modules. When the user saves them, the IDE prompts the user for a file name. A wizard uses a creator object to create such modules.

A creator class implements a creator interface, which inherits from *IOTACreator*. The wizard passes a creator object to the module service's *CreateModule* method, and the IDE calls back to the creator object for the parameters it needs to create the module.

For example, a form wizard that creates a new form typically implements *GetExisting* to return *false* and *GetUnnamed* to return *true*. This creates a module that has no name (so the user must pick a name before the file can be saved) and is not backed by an existing file (so the user must save the file even if the user does not make any changes). Other methods of the creator tell the IDE what kind of file is being created (e.g., project, unit, or form), provide the contents of the file, or return the form name, ancestor name, and other important information. Additional callbacks let a wizard add modules to a newly created project, or add components to a newly created form.

To create a new file, which is often required in a form or project wizard, you usually need to provide the contents of the new file. To do so, write a new class that implements the *IOTAFile* interface. If your wizard can make do with the default file contents, you can return **nil** from any function that returns *IOTAFile*.

For example, suppose your organization has a standard comment block that must appear at the top of each source file. You could do this with a static template in the Object Repository, but the comment block would need to be updated manually to reflect the author and creation date. Instead, you can use a creator to dynamically fill in the comment block when the file is created.

The first step is to write a wizard that creates new units and forms. Most of the creator's functions return zero, empty strings, or other default values, which tells the Tools API to use its default behavior for creating a new unit or form. Override *GetCreatorType* to inform the Tools API what kind of module to create: a unit or a form. To create a unit, return sUnit. To create a form, return sForm. To simplify the code, use a single class that takes the creator type as an argument to the constructor. Save the creator type in a data member, so that *GetCreatorType* can return its value. Implement *NewImplSource* and *NewIntfSource* to return the desired file contents.

```
TCreator = class(TInterfacedObject, IOTAModuleCreator)
public
constructor Create(const CreatorType: string);
  { IOTAModuleCreator }
  function GetAncestorName: string;
  function GetImplFileName: string;
  function GetIntfFileName: string;
  function GetFormName: string;
  function GetMainForm: Boolean;
  function GetShowForm: Boolean;
  function GetShowSource: Boolean;
  function NewFormFile(const FormIdent, AncestorIdent: string): IOTAFile;
  function NewImplSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
  function NewIntfSource(const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
  procedure FormCreated(const FormEditor: IOTAFormEditor);
  { IOTACreator }
  function GetCreatorType: string;
  function GetExisting: Boolean;
  function GetFileSystem: string;
  function GetOwner: IOTAModule;
  function GetUnnamed: Boolean;
private
  FCreatorType: string;
end;
class PACKAGE Creator : public IOTAModuleCreator {
public:
__fastcall Creator(const AnsiString creator_type)
: ref_count(0), creator_type(creator_type) {}
virtual __fastcall ~Creator();
// IOTAModuleCreator
virtual AnsiString __fastcall GetAncestorName();
virtual AnsiString __fastcall GetImplFileName();
virtual AnsiString __fastcall GetIntfFileName();
virtual AnsiString __fastcall GetFormName();
virtual bool __fastcall GetMainForm();
virtual bool __fastcall GetShowForm();
virtual bool __fastcall GetShowSource();
virtual _di_IOTAFile __fastcall NewFormFile(
const AnsiString FormIdent, const AnsiString AncestorIdent);
virtual _di_IOTAFile __fastcall NewImplSource(
```

```
const AnsiString ModuleIdent, const AnsiString FormIdent,
const AnsiString AncestorIdent);
virtual _di_IOTAFile __fastcall NewIntfSource(
const AnsiString ModuleIdent, const AnsiString FormIdent,
const AnsiString AncestorIdent);
virtual void __fastcall FormCreated(
const _di_IOTAFormEditor FormEditor);
// IOTACreator
virtual AnsiString __fastcall GetCreatorType();
 virtual bool __fastcall GetExisting();
virtual AnsiString __fastcall GetFileSystem();
virtual _di_IOTAModule __fastcall GetOwner();
virtual bool __fastcall GetUnnamed();
protected:
// IInterface
virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
virtual ULONG __stdcall AddRef();
virtual ULONG __stdcall Release();
private:
long ref_count;
const AnsiString creator_type;
};
```

Most of the members of *T*Creator return zero, **nil**, or empty strings. The boolean methods return *true*, except *GetExisting*, which returns *false*. The most interesting method is *GetOwner*, which returns a pointer to the current project module, or **nil** if there is no project. There is no simple way to discover the current project or the current project group. Instead, *GetOwner* must iterate over all open modules. If a project group is found, it must be the only project group open, so *GetOwner* returns its current project. Otherwise, the function returns the first project module it finds, or **nil** if no projects are open.

```
function TCreator.GetOwner: IOTAModule;
var
  I: Integer;
  Svc: IOTAModuleServices;
  Module: IOTAModule;
  Project: IOTAProject;
  Group: IOTAProjectGroup;
begin
  { Return the current project. }
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  Result := nil;
  for I := 0 to Svc.ModuleCount - 1 do
  begin
    Module := Svc.Modules[I];
    if Supports(Module, IOTAProject, Project) then
    begin
      { Remember the first project module}
      if Result = nil then
        Result := Project;
    end
    else if Supports(Module, IOTAProjectGroup, Group) then
    begin
      { Found the project group, so return its active project}
      Result := Group.ActiveProject;
      Exit;
    end;
  end;
end;
_di_IOTAModule __fastcall Creator::GetOwner()
{
// Return the current project.
_di_IOTAProject result = 0;
  _di_IOTAModuleServices svc = interface_cast<IOTAModuleServices>(BorlandIDEServices);
  for (int i = 0; i < svc->ModuleCount; ++i)
  begin
    _di_IOTAModule module = svc->Modules[i];
```

```
_di_IOTAProject project;
_di_IOTAProjectGroup group;
if (module->Supports(project)) {
// Remember the first project module.
if (result == 0)
result = project;
} else if (module->Supports(group)) {
// Found the project group, so return its active project.
result = group->ActiveProject;
break;
}
}
return result;
}
```

The creator returns **nil** from *NewFormSource*, to generate a default form file. The interesting methods are *NewImplSource* and *NewIntfSource*, which create an *IOTAFile* instance that returns the file contents.

The *TFile* class implements the *IOTAFile* interface. It returns -1 as the file age (which means the file does not exist), and returns the file contents as a string. To keep the *TFile* class simple, the creator generates the string, and the *TFile* class simply passes it on.

```
TFile = class(TInterfacedObject, IOTAFile)
public
constructor Create(const Source: string);
  function GetSource: string;
  function GetAge: TDateTime;
private
  FSource: string;
end;
constructor TFile.Create(const Source: string);
begin
  FSource := Source;
end;
  function TFile.GetSource: string;
begin
  Result := FSource;
end;
  function TFile.GetAge: TDateTime;
begin
  Result := TDateTime(-1);
end;
class File : public IOTAFile {
public:
__fastcall File(const AnsiString source);
virtual __fastcall ~File();
AnsiString __fastcall GetSource();
System::TDateTime __fastcall GetAge();
protected:
// IInterface
virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
virtual ULONG __stdcall AddRef();
virtual ULONG __stdcall Release();
private:
long ref_count;
AnsiString source;
};
__fastcall File::File(const AnsiString source)
: ref_count(0), source(source)
{}
AnsiString __fastcall File::GetSource()
{
return source;
}
System::TDateTime __fastcall File::GetAge()
{
```

```
return -1;
}
```

You can store the text for the file contents in a resource to make it easier to modify, but for the sake of simplicity, this example hardcodes the source code in the wizard. The example below generates the source code, assuming there is a form. You can easily add the simpler case of a plain unit. Test *FormIdent*, and if it is empty, create a plain unit; otherwise create a form unit. The basic skeleton for the code is the same as the IDE's default (with the addition of the comments at the top, of course), but you can modify it any way you desire.

```
function TCreator.NewImplSource(
                    const ModuleIdent, FormIdent, AncestorIdent: string): IOTAFile;
var
  FormSource: string;
begin
  FormSource :=
  '{ ------------------------------------------------------------------ ' +   #13#10 +
  '%s - description'+   #13#10 +
  'Copyright ©  %y Your company, inc.'+   #13#10 +
  'Created on %d'+   #13#10 +
  'By %u'+   #13#10 +
  ' ------------------------------------------------------------------ }' +   #13#10 +   #13#10;
  return TFile.Create(Format(FormSource, ModuleIdent, FormIdent,
AncestorIdent));
}
_di_IOTAFile __fastcall Creator::NewImplSource(
const AnsiString ModuleIdent,
const AnsiString FormIdent,
const AnsiString AncestorIdent)
{
const AnsiString form_source =
"/*------------------------------------------------------------------\n"
" %m - description\n"
" Copyright \xa9  %y Your company, inc.\n"
" Created on %d\n"
" By %u\n"
"  ------------------------------------------------------------------*/\n"
"\n"
"#include <vcl.h>\n"
"#pragma hdrstop\n"
"\n"
"#include \"%m.h\"\n"
"//------------------------------------------------------------------\n"
"#pragma package(smart_init)\n"
"#pragma resource \"*.dfm\"\n"
"T%f *%f;\n"
"//------------------------------------------------------------------\n"
"__fastcall T%m::T%m(TComponent* Owner)\n"
"        : T%a(Owner)\n"
"{\n"
"}\n"
"//------------------------------------------------------------------\n";
  return new File(expand(form_source, ModuleIdent, FormIdent,
AncestorIdent));
}
```

The final step is to create two form wizards: one uses sUnit as the creator type, and the other uses sForm. As an added benefit for the user, you can use *INTAServices* to add a menu item to the **File ▸ New** menu to invoke each wizard. The menu item's *OnClick* event handler can call the wizard's *Execute* function.

Some wizards need to enable or disable the menu items, depending on what else is happening in the IDE. For example, a wizard that checks a project into a source code control system should disable its Check In menu item if no files are open in the IDE. You can add this capability to your wizard by using notifiers (see page 1289).

See Also

Working with Files and Editors (see page 1297)

3.2.1.6.4 Debugging a Wizard

The Tools API provides you with a lot of flexibility in how your wizard interacts with the IDE. With the flexibility comes responsibility, however. It is easy to wind up with dangling pointers or other access violations.

When writing wizards that use the native tools API, you can write code that causes the IDE to crash. It is also possible that you write a wizard that installs but does not act the way you want it to. One of the challenges of working with design-time code is debugging. It's an easy problem to solve, however. Because the wizard is installed in Delphi itself, you simply need to set the package's Host Application to the Delphi executable from the **Run ▸ Parameters**... menu item.

When you want (or need) to debug the package, don't install it. Instead, choose **Run ▸ Run** from the menu bar. This starts up a new instance of Delphi. In the new instance, install the already-compiled package by choosing **Components ▸ Install Package**... from the menu bar. Back in the original instance of Delphi, you should now see the telltale blue dots that tell you where you can set breakpoints in the wizard source code. (If not, double-check your compiler options to be sure you enabled debugging; make sure you loaded the right package; and double-check the process modules to make extra sure that you loaded the .bpl file you wanted to load.)

You cannot debug into the VCL or RTL code this way, but you have full debug capabilities for the wizard itself, which might be enough to tell what is going wrong.

See Also

Using Native IDE Objects (see page 1297)

3.2.1.6.5 Deleting Toolbar Buttons

There is no convenient function for removing a button from a toolbar; you must send the CM_CONTROLCHANGE message, where the first parameter is the control to change, and the second parameter is zero to remove it or non-zero to add it to the toolbar. After removing the toolbar buttons, the destructor deletes the action and menu item. Deleting these items automatically removes them from the IDE's ActionList and MainMenu.

```
procedure remove_action (Action: TAction; ToolBar: TToolBar);
var
  I: Integer;
  Btn: TToolButton;
begin
  for I := ToolBar.ButtonCount - 1 downto 0 do
  begin
    Btn := ToolBar.Buttons[I];
    if Btn.Action = Action then
    begin
      { Remove "Btn" from "ToolBar" }
      ToolBar.Perform(CM_CONTROLCHANGE, WPARAM(Btn), 0);
      Btn.Free;
    end;
  end;
end;
destructor MyWizard.Destroy;
var
  Services: INTAServices;
  Btn: TToolButton;
begin
  Supports(BorlandIDEServices, INTAServices, Services);
  { Check all the toolbars, and remove any buttons that use this action. }
remove_action(NewAction, Services.ToolBar[sCustomToolBar]);
```

```
remove_action(NewAction, Services.ToolBar[sDesktopToolBar]);
remove_action(NewAction, Services.ToolBar[sStandardToolBar]);
remove_action(NewAction, Services.ToolBar[sDebugToolBar]);
remove_action(NewAction, Services.ToolBar[sViewToolBar]);
remove_action(NewAction, Services.ToolBar[sInternetToolBar]);
  NewItem.Free;
  NewAction.Free;
end;
void __fastcall remove_action (TAction* action, TToolBar* toolbar)
{
for (int i = toolbar->ButtonCount; --i >= 0; )
{
TToolButton* button = toolbar->Buttons[i];
if (button->Action == action)
{
// Remove "button" from "toolbar".
toolbar->Perform(CM_CONTROLCHANGE, WPARAM(button), 0);
delete button;
}
}
}
__fastcall MyWizard::~MyWizard()
{
_di_INTAServices services;
  BorlandIDEServices->Supports(services);
  // Check all the toolbars, and remove any buttons that use
// this action.
remove_action(action, services->ToolBar[sCustomToolBar]);
remove_action(action, services->ToolBar[sDesktopToolBar]);
remove_action(action, services->ToolBar[sStandardToolBar]);
remove_action(action, services->ToolBar[sDebugToolBar]);
remove_action(action, services->ToolBar[sViewToolBar]);
remove_action(action, services->ToolBar[sInternetToolBar]);
  delete menu_item;
delete action;
}
```

See Also

Adding an Image to the Image List (see page 1279)

Adding an Action to the Action List (see page 1278)

3.2.1.6.6 Extending the IDE

You can extend and customize the IDE with your own menu items, tool bar buttons, dynamic form-creation wizards, and more, using the Open Tools API (often shortened to just Tools API). The Tools API is a suite of over 100 interfaces that interact with and control the IDE, including the main menu, the tool bars, the main action list and image list, the source editor's internal buffers, keyboard macros and bindings, forms and their components in the form editor, the debugger and the process being debugged, code completion, the message view, and the To-Do list.

Using the Tools API is simply a matter of writing classes that implement certain interfaces, and calling on services provided by other interfaces. Your Tools API code must be compiled and loaded into the IDE at design-time as a design-time package or in a DLL. Thus, writing a Tools API extension is somewhat like writing a property or component editor. Before tackling this material, make sure you are familiar with the basics of working with packages (see page 2211) and registering components (see page 1350).

Note: To enable the use of the Tools API, you must link with runtime packages and ensure that “designide” is listed as one of the packages you are linking with. In the IDE, you can do this on the Project->Options->Packages

dialog box. If you are building from the command line, make sure that `-LUdesignide` is passed to dcc32.exe. The following topics describe how to use the Tools API:

- Overview of the Tools API (see page 1294)
- Writing a wizard class (see page 1297)
- Obtaining Tools API services (see page 1293)
- Working with files and editors (see page 1297)
- Creating forms and projects (see page 1279)
- Notifying a wizard of IDE events (see page 1289)

See Also

Making Components Available at Design Time: Overview (see page 1350)

3.2.1.6.7 Implementing the Wizard Interfaces

Every wizard class must implement at least *IOTAWizard*, which requires implementing its ancestors, too: *IOTANotifier* and *IInterface*. Form and project wizards must implement all their ancestor interfaces, namely, *IOTARepositoryWizard*, *IOTAWizard*, *IOTANotifier*, and *IInterface*.

For C++, to use NotifierObject as a base class you must use multiple inheritance. Your wizard class must inherit from NotifierObject and from the wizard interfaces that you need to implement, such as IOTAWizard. Because IOTAWizard inherits from IOTANotifier and IInterface, there is an ambiguity in the derived class: functions such as AddRef() are declared in every branch of the ancestral inheritance graph. To resolve this problem, pick one base class as the primary base class and delegate all ambiguous functions to that one class. For example, the class declaration might look as follows:

```
class PACKAGE MyWizard : public NotifierObject, public IOTAMenuWizard {
    typedef NotifierObject inherited;
    public:
        // IOTAWizard
        virtual AnsiString __fastcall GetIDString();
        virtual AnsiString __fastcall GetName();
        virtual TWizardState __fastcall GetState();
        virtual void __fastcall Execute();
        // IOTAMenuWizard
        virtual AnsiString __fastcall GetMenuText();
      void __fastcall AfterSave();
        void __fastcall BeforeSave();
        void __fastcall Destroyed();
        void __fastcall Modified();
    protected:
        // IInterface
        virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
        virtual ULONG __stdcall AddRef();
        virtual ULONG __stdcall Release();
};
// implementation
ULONG __stdcall MyWizard::AddRef()  { return inherited::AddRef(); }
ULONG __stdcall MyWizard::Release() { return inherited::Release(); }
HRESULT __stdcall MyWizard::QueryInterface(const GUID& iid, void** obj)
{
    if (iid == __uuidof(IOTAMenuWizard)) {
        *obj = static_cast<IOTAMenuWizard*>(this);
        static_cast<IOTAMenuWizard*>(*obj)->AddRef();
        return S_OK;
    }
    if (iid == __uuidof(IOTAWizard)) {
        *obj = static_cast<IOTAWizard*>(this);
        static_cast<IOTAWizard*>(*obj)->AddRef();
        return S_OK;
    }
    return inherited::QueryInterface(iid, obj);
}
```

Your implementation of *IInterface* must follow the normal rules for Delphi interfaces, which are the same as the rules for COM interfaces. That is, *QueryInterface* performs type casts, and *_AddRef* and *_Release* manage reference counting. You might want to use a common base class to simplify writing wizard and notifier classes. For this purpose, the ToolsAPI unit defines a class, *TNotifierObject*, which implements *IOTANotifier* interface with empty method bodies.

You can write a class similar to *TNotifierObject* in C++.

```
class PACKAGE NotifierObject : public IOTANotifier {
    public:
        __fastcall NotifierObject() : ref_count(0) {}
        virtual __fastcall ~NotifierObject();
        void __fastcall AfterSave();
        void __fastcall BeforeSave();
        void __fastcall Destroyed();
        void __fastcall Modified();
    protected:
        // IInterface
        virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
        virtual ULONG __stdcall AddRef();
        virtual ULONG __stdcall Release();
    private:
        long ref_count;
};
// implementation
ULONG __stdcall NotifierObject::AddRef()
{
    return InterlockedIncrement(&ref_count);
}
ULONG __stdcall NotifierObject::Release()
{
    ULONG result = InterlockedDecrement(&ref_count);
    if (ref_count == 0)
        delete this;
    return result;
}
HRESULT __stdcall NotifierObject::QueryInterface(const GUID& iid, void** obj)
{
    if (iid == __uuidof(IInterface)) {
        *obj = static_cast<IInterface*>(this);
        static_cast<IInterface*>(*obj)->AddRef();
        return S_OK;
    }
    if (iid == __uuidof(IOTANotifier)) {
        *obj = static_cast<IOTANotifier*>(this);
        static_cast<IOTANotifier*>(*obj)->AddRef();
        return S_OK;
    }
    return E_NOINTERFACE;
}
```

Although wizards inherit from *IOTANotifier*, and must therefore implement all of its functions, the IDE does not usually make use of those functions, so your implementations can be empty (as they are in *TNotifierObject*). Thus, when you write your wizard class, you need only declare and implement those interface methods introduced by the wizard interfaces, accepting the *TNotifierObject* implementation of *IOTANotifier*.

```
// C++ empty implementations
void __fastcall NotifierObject::AfterSave()  {}
void __fastcall NotifierObject::BeforeSave() {}
void __fastcall NotifierObject::Destroyed()  {}
void __fastcall NotifierObject::Modified()   {}
```

See Also

Installing the Wizard Package (see page 1288)

3.2.1.6.8 Installing the Wizard Package

As with any other design-time package, a wizard package must have a *Register* function. (See Registering components (see page 1352) for details about the *Register* function.) In the *Register* function, you can register any number of wizards by calling *RegisterPackageWizard*, and passing a wizard object as the sole argument, as shown below:

```
procedure Register;
begin
  RegisterPackageWizard(MyWizard.Create);
  RegisterPackageWizard(MyOtherWizard.Create);
end;
```

```
namespace Example {
    void __fastcall PACKAGE Register()
    {
        RegisterPackageWizard(new MyWizard());
        RegisterPackageWizard(new MyOtherWizard());
    }
}
```

You can also register property editors, components, and so on, as part of the same package.

Remember that a design-time package is part of the main RAD Studio application, which means any form names must be unique throughout the entire application and all other design-time packages. This is the main disadvantage to using packages: you never know what someone else might name their forms.

During development, install the wizard package the way you would any other design-time package: click the Install button in the package manager. The IDE will compile and link the package and attempt to load it. The IDE displays a dialog box telling you whether it successfully loaded the package.

See Also

Implementing the Wizard Interfaces (see page 1286)

3.2.1.6.9 Interface Version Numbers

If you look closely at the declarations of some of the interfaces, such as *IOTAMessageServices*, you will see that they inherit from other interfaces with similar names, such as *IOTAMessageServices50*, which inherits from *IOTAMessageServices40*. This use of version numbers helps insulate your code from changes between releases of Delphi.

The Tools API follows the basic principle of COM, namely, that an interface and its GUID never change. If a new release adds features to an interface, the Tools API declares a new interface that inherits from the old one. The GUID remains the same, attached to the old, unchanged interface. The new interface gets a brand new GUID. Old wizards that use the old GUIDs continue to work.

The Tools API also changes interface names to try to preserve source-code compatibility. To see how this works, it is important to distinguish between the two kinds of interfaces in the Tools API: Borland-implemented and user-implemented. If the IDE implements the interface, the name stays with the most recent version of the interface. The new functionality does not affect existing code. The old interfaces have the old version number appended.

For a user-implemented interface, however, new member functions in the base interface require new functions in your code. Therefore, the name tends to stick with the old interface, and the new interface has a version number tacked onto the end.

For example, consider the message services. Delphi 6 introduced a new feature: message groups. Therefore, the basic message services interface required new member functions. These functions were declared in a new interface class, which retained the name *IOTAMessageServices*. The old message services interface was renamed to *IOTAMessageServices50* (for version 5). The GUID of the old *IOTAMessageServices* is the same as the GUID of the new *IOTAMessageServices50* because the member functions are the same.

Consider *IOTAIDENotifier* as an example of a user-implemented interface. Delphi 5 added new overloaded functions: *AfterCompile* and *BeforeCompile*. Existing code that used *IOTAIDENotifier* did not need to change, but new code that required the new functionality had to be modified to override the new functions inherited from *IOTAIDENotifier50*. Version 6 did not add any more functions, so the current version to use is *IOTAIDENotifier50*.

The rule of thumb is to use the most-derived class when writing new code. Leave the source code alone if you are merely recompiling an existing wizard under a new release of Delphi.

See Also

Obtaining Tools API Services (see page 1293)

3.2.1.6.10 Notifying a Wizard of IDE Events

An important aspect of writing a well-behaved wizard is to have the wizard respond to IDE events. In particular, any wizard that keeps track of module interfaces must know when the user closes the module, so the wizard can release the interface. To do this, the wizard needs a notifier, which means you must write a notifier class.

All notifier classes implement one or more notifier interfaces. The notifier interfaces define callback methods; the wizard registers a notifier object with the Tools API, and the IDE calls back to the notifier when something important happens.

Every notifier interface inherits from *IOTANotifier*, although not all of its methods are used for a particular notifier. The following table lists all the notifier interfaces, and gives a brief description of each one.

Notifier interfaces

Interface	Description
IOTANotifier	Abstract base class for all notifiers
IOTABreakpointNotifier	Triggering or changing a breakpoint in the debugger
IOTADebuggerNotifier	Running a program in the debugger, or adding or deleting breakpoints
IOTAEditLineNotifier	Tracking movements of lines in the source editor
IOTAEditorNotifier	Modifying or saving a source file, or switching files in the editor
IOTAFormNotifier	Saving a form, or modifying the form or any components on the form (or data module)
IOTAIDENotifier	Loading projects, installing packages, and other global IDE events
IOTAMessageNotifier	Adding or removing tabs (message groups) in the message view
IOTAModuleNotifier	Changing, saving, or renaming a module
IOTAProcessModNotifier	Loading a process module in the debugger
IOTAProcessNotifier	Creating or destroying threads and processes in the debugger
IOTAThreadNotifier	Changing a thread's state in the debugger
IOTAToolsFilterNotifier	Invoking a tools filter

To see how to use notifiers, consider the example in Creating forms and projects (see page 1279). Using module creators, the example creates a wizard that adds a comment to each source file. The comment includes the unit's initial name, but the user almost always saves the file under a different name. In that case, it would be a courtesy to the user if the wizard updated the comment to match the file's true name.

To do this, you need a module notifier. The wizard saves the module interface that *CreateModule* returns, and uses it to register a module notifier. The module notifier receives notification when the user modifies the file or saves the file, but these events are not important for this wizard, so the *AfterSave* and related functions all have empty bodies. The important function is *ModuleRenamed*, which the IDE calls when the user saves the file under a new name. The declaration for the module notifier class is shown below:

```
TModuleIdentifier = class(TNotifierObject, IOTAModuleNotifier)
public
constructor Create(const Module: IOTAModule);
  destructor Destroy; override;
  function CheckOverwrite: Boolean;
  procedure ModuleRenamed(const NewName: string);
  procedure Destroyed;
private
  FModule: IOTAModule;
  FName: string;
  FIndex: Integer;
end;
class ModuleNotifier : public NotifierObject, public IOTAModuleNotifier
{
typedef NotifierObject inherited;
public:
__fastcall ModuleNotifier(const _di_IOTAModule module);
__fastcall ~ModuleNotifier();
// IOTAModuleNotifier
virtual bool __fastcall CheckOverwrite();
virtual void __fastcall ModuleRenamed(const AnsiString NewName);
// IOTANotifier
void __fastcall AfterSave();
void __fastcall BeforeSave();
void __fastcall Destroyed();
void __fastcall Modified();
protected:
// IInterface
virtual HRESULT __stdcall QueryInterface(const GUID&, void**);
virtual ULONG __stdcall AddRef();
virtual ULONG __stdcall Release();
private:
_di_IOTAModule module;
AnsiString name;        // Remember the module's old name.
int index;              // Notifier index.
};
```

One way to write a notifier is to have it register itself automatically in its constructor. The destructor unregisters the notifier. In the case of a module notifier, the IDE calls the *Destroyed* method when the user closes the file. In that case, the notifier must unregister itself and release its reference to the module interface. The IDE releases its reference to the notifier, which reduces its reference count to zero and frees the object. Therefore, you need to write the destructor defensively: the notifier might already be unregistered.

```
constructor TModuleNotifier.Create( const Module: IOTAModule);
begin
  FIndex := -1;
  FModule := Module;
  { Register this notifier. }
  FIndex := Module.AddNotifier(self);
  { Remember the module's old name. }
  FName := ChangeFileExt(ExtractFileName(Module.FileName), '');
end;
destructor TModuleNotifier.Destroy;
begin
  { Unregister the notifier if that hasn't happened already. }
  if Findex >= 0 then
    FModule.RemoveNotifier(FIndex);
end;
procedure TModuleNotifier.Destroyed;
begin
   { The module interface is being destroyed, so clean up the notifier. }
  if Findex >= 0 then
  begin
    { Unregister the notifier. }
    FModule.RemoveNotifier(FIndex);
    FIndex := -1;
```

```
  end;
  FModule := nil;
end;
__fastcall ModuleNotifier::ModuleNotifier(const _di_IOTAModule module)
: index(-1), module(module)
{
// Register this notifier.
index = module->AddNotifier(this);
// Remember the module's old name.
name = ChangeFileExt(ExtractFileName(module->FileName), "");
}
__fastcall ModuleNotifier::~ModuleNotifier()
{
// Unregister the notifier if that hasn't happened already.
if (index >= 0)
module->RemoveNotifier(index);
}
void __fastcall ModuleNotifier::Destroyed()
{
// The module interface is being destroyed, so clean up the notifier.
if (index >= 0)
{
// Unregister the notifier.
module->RemoveNotifier(index);
index = -1;
}
module = 0;
}
```

The IDE calls back to the notifier's *ModuleRenamed* function when the user renames the file. The function takes the new name as a parameter, which the wizard uses to update the comment in the file. To edit the source buffer, the wizard uses an edit position interface. The wizard finds the right position, double checks that it found the right text, and replaces that text with the new name.

```
procedure TModuleNotifier.ModuleRenamed(const NewName: string);
var
  ModuleName: string;
  I: Integer;
  Editor: IOTAEditor;
  Buffer: IOTAEditBuffer;
  Pos: IOTAEditPosition;
  Check: string;
begin
  { Get the module name from the new file name. }
  ModuleName := ChangeFileExt(ExtractFileName(NewName), '');
for I := 0 to FModule.GetModuleFileCount - 1 do
  begin
   { Update every source editor buffer. }
    Editor := FModule.GetModuleFileEditor(I);
    if Supports(Editor, IOTAEditBuffer, Buffer) then
    begin
      Pos := Buffer.GetEditPosition;
{ The module name is on line 2 of the comment.
        Skip leading white space and copy the old module name,
        to double check we have the right spot. }
      Pos.Move(2, 1);
      Pos.MoveCursor(mmSkipWhite or mmSkipRight);
Check := Pos.RipText('', rfIncludeNumericChars or rfIncludeAlphaChars);
if Check = FName then
      begin
        Pos.Delete(Length(Check));    // Delete the old name.
        Pos.InsertText(ModuleName);   // Insert the new name.
        FName := ModuleName;          // Remember the new name.
      end;
    end;
  end;
```

```
end;
void __fastcall ModuleNotifier::ModuleRenamed(const AnsiString NewName)
{
// Get the module name from the new file name.
AnsiString ModuleName = ChangeFileExt(ExtractFileName(NewName), "");
for (int i = 0; i < module->GetModuleFileCount(); ++i)
{
// Update every source editor buffer.
_di_IOTAEditor editor = module->GetModuleFileEditor(i);
_di_IOTAEditBuffer buffer;
if (editor->Supports(buffer))
{
_di_IOTAEditPosition pos = buffer->GetEditPosition();
// The module name is on line 2 of the comment.
// Skip leading white space and copy the old module name,
// to double check we have the right spot.
pos->Move(2, 1);
pos->MoveCursor(mmSkipWhite | mmSkipRight);
AnsiString check = pos->RipText("", rfIncludeNumericChars | rfIncludeAlphaChars);
if (check == name)
{
pos->Delete(check.Length());    // Delete the old name.
pos->InsertText(ModuleName);    // Insert the new name.
name = ModuleName;              // Remember the new name.
}
}
}
}
```

What if the user inserts additional comments above the module name? In that case, you need to use an edit line notifier to keep track of the line number where the module name sits. To do this, use the *IOTAEditLineNotifier* and *IOTAEditLineTracker* interfaces.

You need to be cautious when writing notifiers. You must make sure that no notifier outlives its wizard. For example, if the user were to use the wizard to create a new unit, then unload the wizard, there would still be a notifier attached to the unit. The results would be unpredictable, but most likely, the IDE would crash. Thus, the wizard needs to keep track of all of its notifiers, and must unregister every notifier before the wizard is destroyed. On the other hand, if the user closes the file first, the module notifier receives a *Destroyed* notification, which means the notifier must unregister itself and release all references to the module. The notifier must remove itself from the wizard's master notifier list, too.

Below is the final version of the wizard's *Execute* function. It creates the new module, uses the module interface and creates a module notifier, then saves the module notifier in an interface list (*TInterfaceList*).

```
procedure DocWizard.Execute;
var
  Svc: IOTAModuleServices;
  Module: IOTAModule;
  Notifier: IOTAModuleNotifier;
begin
  { Return the current project. }
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  Module := Svc.CreateModule(TCreator.Create(creator_type));
  Notifier := TModuleNotifier.Create(Module);
list.Add(Notifier);
end
void __fastcall DocWizard::Execute()
{
_di_IOTAModuleServices svc;
  BorlandIDEServices->Supports(svc);
_di_IOTAModule module = svc->CreateModule(new Creator(creator_type));
_di_IOTAModuleNotifier notifier = new ModuleNotifier(module);
list->Add(notifier);
}
```

The wizard's destructor iterates over the interface list and unregisters every notifier in the list. Simply letting the interface list

release the interfaces it holds is not sufficient because the IDE also holds the same interfaces. You must tell the IDE to release the notifier interfaces in order to free the notifier objects. In this case, the destructor tricks the notifiers into thinking their modules have been destroyed. In a more complicated situation, you might find it best to write a separate Unregister function for the notifier class.

```
destructor DocWizard.Destroy; override;
var
  Notifier: IOTAModuleNotifier;
  I: Integer;
begin
  { Unregister all the notifiers in the list. }
for I := list.Count - 1 downto 0 do
  begin
    Supports(list.Items[I], IOTANotifier, Notifier);
    { Pretend the associated object has been destroyed.
      That convinces the notifier to clean itself up. }
    Notifier.Destroyed;
    list.Delete(I);
  end;
  list.Free;
  FItem.Free;
end;
__fastcall DocWizard::~DocWizard()
{
// Unregister all the notifiers in the list.
for (int i = list->Count; --i >= 0; )
{
_di_IOTANotifier notifier;
    list->Items[i]->Supports(notifier);
// Pretend the associated object has been destroyed.
// That convinces the notifier to clean itself up.
notifier->Destroyed();
list->Delete(i);
}
delete list;
delete item;
}
```

The rest of the wizard manages the mundane details of registering the wizard, installing menu items, and the like.

See Also

Obtaining Tools API Services (see page 1293)

Using Module Interfaces (see page 1296)

3.2.1.6.11 Obtaining Tools API Services

To do anything useful, a wizard needs access to the IDE: its editors, windows, menus, and so on. This is the role of the service interfaces. The Tools API includes many services, such as action services to perform file actions, editor services to access the source code editor, debugger services to access the debugger, and so on. The following table summarizes all the service interfaces.

Tools API service interfaces

Interface	Description
INTAServices	Provides access to native IDE objects: main menu, action list, image list, and tool bars.
IOTAActionServices	Performs basic file actions: open, close, save, and reload a file.
IOTACodeCompletionServices	Provides access to code completion, allowing a wizard to install a custom code completion manager.

IOTADebuggerServices	Provides access to debugger.
IOTAEditorServices	Provides access to source code editor and its internal buffers.
IOTAKeyBindingServices	Permits a wizard to register custom keyboard bindings.
IOTAKeyboardServices	Provides access to keyboard macros and bindings.
IOTAKeyboardDiagnostics	Toggle debugging of keystrokes.
IOTAMessageServices	Provides access to message view.
IOTAModuleServices	Provides access to open files.
IOTAPackageServices	Queries the names of all installed packages and their components.
IOTAServices	Miscellaneous services.
IOTAToDoServices	Provides access to the To-Do list, allowing a wizard to install a custom To-Do manager.
IOTAToolsFilter	Registers tools filter notifiers.
IOTAWizardServices	Registers and unregisters wizards.

To use a service interface, cast the *BorlandIDEServices* variable to the desired service using the global Supports function, which is defined in the SysUtils unit. For example,

```
procedure set_keystroke_debugging(debugging: Boolean);
var
  diag: IOTAKeyboardDiagnostics
begin
  if Supports(BorlandIDEServices, IOTAKeyboardDiagnostics, diag) then
    diag.KeyTracing := debugging;
end;
void set_keystroke_debugging(bool debugging)
{
_di_IOTAKeyboardDiagnostics diag;
if (BorlandIDEServices->Supports(diag))
diag->KeyTracing = debugging;
}
```

If your wizard needs to use a specific service often, you can keep a pointer to the service as a data member of your wizard class.

The following topics discuss special considerations when working with the Tools API service interfaces:

- Using native IDE objects (see page 1297)
- Debugging a wizard (see page 1284)
- Interface version numbers (see page 1288)

See Also

Working with Files and Editors (see page 1297)

Creating Forms and Projects (see page 1279)

Notifying a Wizard of IDE Events (see page 1289)

3.2.1.6.12 Overview of the Tools API

All of the Tools API declarations reside in a single unit, ToolsAPI. To use the Tools API, you typically use the designide package, which means you must build your Tools API add-in as a design-time package or as a DLL that uses runtime packages. For information about package and library issues, see Installing the wizard package (see page 1288).

The main interface for writing a Tools API extension is *IOTAWizard*, so most IDE add-ins are called wizards. C++Builder and Delphi wizards are, for the most part, interoperable. You can write and compile a wizard in Delphi, then use it in C++Builder, and

vice versa. Interoperability works best with the same version number, but it is also possible to write wizards so they can be used in future versions of both products.

To use the Tools API, you write wizard classes that implement one or more of the interfaces defined in the ToolsAPI unit.

A wizard makes use of services that the Tools API provides. Each service is an interface that presents a set of related functions. The implementation of the interface is hidden within the IDE. The Tools API publishes only the interface, which you can use to write your wizards without concerning yourself with the implementation of the interfaces. The various services offer access to the source editor, form designer, debugger, and so on. See Obtaining Tools API services (see page 1293) for information about using the interfaces that expose services to your wizard.

The service and other interfaces fall into two basic categories. You can tell them apart by the prefix used for the type name:

- The NTA (native tools API) grants direct access to actual IDE objects, such as the IDE's *TMainMenu* object. When using these interfaces, the wizard must use CodeGear packages, which also means the wizard is tied to a specific version of the IDE. The wizard can reside in a design-time package or in a DLL that uses runtime packages.
- The OTA (open tools API) does not require packages and accesses the IDE only through interfaces. In theory, you could write a wizard in any language that supports COM-style interfaces, provided you can also work with the Delphi calling conventions and Delphi types such as AnsiString. OTA interfaces do not grant full access to the IDE, but almost all the functionality of the Tools API is available through OTA interfaces. If a wizard uses only OTA interfaces, it is possible to write a DLL that is not dependent on a specific version of the IDE.

The Tools API has two kinds of interfaces: those that you, the programmer, must implement and those that the IDE implements. Most of the interfaces are in the latter category: the interfaces define the capability of the IDE but hide the actual implementation. The kinds of interfaces that you must implement fall into three categories: wizards, notifiers, and creators:

- As mentioned earlier in this topic, a wizard class implements the *IOTAWizard* interface and possibly derived interfaces.
- A notifier is another kind of interface in the Tools API. The IDE uses notifiers to call back to your wizard when something interesting happens. You write a class that implements the notifier interface, register the notifier with the Tools API, and the IDE calls back to your notifier object when the user opens a file, edits source code, modifies a form, starts a debugging session, and so on. Notifiers are covered in Notifying a wizard of IDE events (see page 1289) .
- A creator is another kind of interface that you must implement. The Tools API uses creators to create new units, projects, or other files, or to open existing files. See Creating forms and projects (see page 1279) for information about creator interfaces.

Other important interfaces are modules and editors. A module interface represents an open unit, which has one or more files. An editor interface represents an open file. Different kinds of editor interfaces give you access to different aspects of the IDE: the source editor for source files, the form designer for form files, and project resources for a resource file. See Working with files and editors (see page 1297) for information about module and editor interfaces.

3.2.1.6.13 Using Editor Interfaces

Every module has at least one editor interface. Some modules have several editors, such as a source (.pas) file and form description (.dfm) file. All editors implement the *IOTAEditor* interface; cast the editor to a specific type to learn what kind of editor it is. For example, to obtain the form editor interface for a unit, you can do the following:

```
{ Return the form editor for a module, or nil if the unit has no form. }
function GetFormEditor(Module: IOTAModule): IOTAFormEditor;
var
  I: Integer;
  Editor: IOTAEditor;
begin
  for I := 0 to Module.ModuleFileCount - 1 do
  begin
    Editor := Module.ModuleFileEditors[I];
    if Supports(Editor, IOTAFormEditor, Result) then
      Exit;
  end;
  Result := nil;
end;
```

```
// Return the form editor for a module, or 0 if the unit has no form.
_di_IOTAFormEditor __fastcall GetFormEditor(_di_IOTAModule module)
{
for (int i = 0; i < module->ModuleFileCount; ++i)
{
_di_IOTAEditor editor = module->ModuleFileEditors[i];
_di_IOTAFormEditor formEditor;
if (editor->Supports(formEditor))
return formEditor;
}
return 0;
}
```

The editor interfaces give you access to the editor's internal state. You can examine the source code or components that the user is editing, make changes to the source code, components, or properties, change the selection in the source and form editors, and carry out almost any editor action that the end user can perform.

Using a form editor interface, a wizard can access all the components on the form. Each component (including the root form or data module) has an associated *IOTAComponent* interface. A wizard can examine or change most of the component's properties. If you need complete control over the component, you can cast the *IOTAComponent* interface to *INTAComponent*. The native component interface enables your wizard to access the *TComponent* pointer directly. This is important if you need to read or modify a class-type property, such as *TFont*, which is possible only through NTA-style interfaces.

See Also

Using Module Interfaces (see page 1296)

3.2.1.6.14 Using Module Interfaces

To obtain a module interface, start with the module services (*IOTAModuleServices*). You can query the module services for all open modules, look up a module from a file name or form name, or open a file to obtain its module interface.

There are different kinds of modules for different kinds of files, such as projects, resources, and type libraries. Cast a module interface to a specific kind of module interface to learn whether the module is of that type. For example, one way to obtain the current project group interface is as follows:

```
{ Return the current project group, or nil if there is no project group. }
function CurrentProjectGroup: IOTAProjectGroup;
var
  I: Integer;
  Svc: IOTAModuleServices;
  Module: IOTAModule;
begin
  Supports(BorlandIDEServices, IOTAModuleServices, Svc);
  for I := 0 to Svc.ModuleCount - 1 do
  begin
    Module := Svc.Modules[I];
    if Supports(Module, IOTAProjectGroup, Result) then
      Exit;
  end;
  Result := nil;
end;
// Return the current project group, or 0 if there is no project group.
_di_IOTAProjectGroup __fastcall CurrentProjectGroup()
{
_di_IOTAModuleServices svc;
  BorlandIDEServices->Supports(svc);
  for (int i = 0; i < svc->ModuleCount; ++i)
{
_di_IOTAModule module = svc->Modules[i];
_di_IOTAProjectGroup group;
if (module->Supports(group))
return group;
```

```
}
return 0;
}
```

See Also

Using Editor Interfaces (see page 1295)

3.2.1.6.15 Using Native IDE Objects

Wizards have full access to the main menu, tool bars, action list, and image list of the IDE. (Note that the IDE's many context menus are not accessible through the Tools API.)

The starting point for working with native IDE objects is the *INTAServices* interface. Use this interface to add an image to the image list, an action to the action list, a menu item to the main menu, and a button to a tool bar. You can tie the action to the menu item and tool button. When the wizard is destroyed, it must clean up the objects it creates, but it must not delete the image it added to the image list. Deleting an image would scramble the indices for all images added after this wizard.

The wizard uses the actual *TMainMenu*, *TActionList*, *TImageList*, and *TToolBar* objects from the IDE, so you can write code the way you would any other application. It also means you have a lot of scope for crashing the IDE or otherwise disabling important features, such as deleting the File menu. Debugging a wizard (see page 1284) discusses steps you can take to debug your wizard if you find it has caused problems like these.

The following topics illustrate how to perform these tasks:

- Adding an image to the image list (see page 1279)
- Adding an action to the action list (see page 1278)
- Deleting toolbar buttons (see page 1284)

3.2.1.6.16 Working with Files and Editors

It is important to understand how the Tools API works with files. The main interface is *IOTAModule*. A module represents a set of logically related open files. For example, a single module represents a single unit. The module, in turn, has one or more editors, where each editor represents one file, such as the unit source (.pas) or form (.dfm or .xfm) file. The editor interfaces reflect the internal state of the IDE's editors, so a wizard can see the modified code and forms that the user sees, even if the user has not saved any changes.

The following topics provide information about the module and editor interfaces:

- Using module interfaces (see page 1296)
- Using editor interfaces (see page 1295)

See Also

Obtaining Tools API Services (see page 1293)

Creating Forms and Projects (see page 1279)

Notifying a Wizard of IDE Events (see page 1289)

3.2.1.6.17 Writing a Wizard Class

There are four kinds of wizards, where the wizard kind depends on the interfaces that the wizard class implements. The following table describes the four kinds of wizards.

The four kinds of wizards

Interface	Description
IOTAFormWizard	Typically creates a new unit, form, or other file
IOTAMenuWizard	Automatically added to Help menu
IOTAProjectWizard	Typically creates a new application or other project
IOTAWizard	Miscellaneous wizard that doesn't fit into other categories

The four kinds of wizards differ only in how the user invokes the wizard:

- A menu wizard is added to the IDE's Help menu. When the user picks the menu item, the IDE calls the wizard's *Execute* function. Plain wizards offer much more flexibility, so menu wizards are typically used only for prototypes and debugging.
- Form and project wizards are called repository wizards because they reside in the Object Repository. The user invokes these wizards from the New Items dialog box. The user can also see the wizards in the object repository (by choosing the **Tools** ▶ **Repository** menu item). The user can check the New Form check box for a form wizard, which tells the IDE to invoke the form wizard when the user chooses the **File** ▶ **New** ▶ **Form** menu item. The user can also check the Main Form check box. This tells the IDE to use the form wizard as the default form for a new application. The user can check the New Project check box for a project wizard. When the user chooses **File** ▶ **New** ▶ **Application**, the IDE invokes the selected project wizard.
- The fourth kind of wizard is for situations that don't fit into the other categories. A plain wizard does not do anything automatically or by itself. Instead, you must define how the wizard is invoked.

The Tools API does not enforce any restrictions on wizards, such as requiring a project wizard to create a project. You can just as easily write a project wizard to create a form and a form wizard to create a project (if that's something you really want to do).

The following topics provide details on how to implement and install a wizard:

- Implementing the wizard interfaces (see page 1286)
- Installing the wizard package (see page 1288)

3.2.1.7 Handling messages

Topics

Name	Description
Broadcasting a Message to All Controls in a Form (see page 1301)	When your component changes global settings that affect all of the controls in a form or other container, you may want to send a message to those controls so that they can update themselves appropriately. Not every control may need to respond to the notification, but by broadcasting the message, you can inform all controls that know how to respond and allow the other controls to ignore the message. To broadcast a message to all the controls in another control, use the Broadcast method. Before you broadcast a message, you fill out a message record (see page 1305) with the information you want... more (see page 1301)
Calling a Control's Message Handler Directly (see page 1302)	Sometimes there is only a single control that needs to respond to your message. If you know the control that should receive your message, the simplest and most straightforward way to send the message is to call the control's Perform method. There are two main reasons why you call a control's *Perform* method: • You want to trigger the same response that the control makes to a standard Windows (or other) message. For example, when a grid control receives a keystroke message, it creates an inline edit control and then sends the keystroke message on to the edit control. • You may... more (see page 1302)

Responding to System Events (see page 1302)	When the widget layer receives an event notification from the operating system, it generates a special event object (*QEvent* or one of its descendants) to represent the event. The event object contains read-only information about the event that occurred. The type of the event object indicates the type of event that occurred. The widget layer notifies your CLX component of system events using a special signal of type event. It passes the *QEvent* object to the signal handler for the event. The processing of the event signal is a bit more complicated than processing other signals because it goes... more (see page 1302)
Sending a Message That Does Not Execute Immediately (see page 1303)	There are times you may want to send a message but you do not know whether it is safe for the target of the message to execute right away. For example, if the code that sends a message is called from an event handler on the target control, you may want to make sure that the event handler has finished executing before the control executes your message. You can handle this situation as long as you do not need to know the message result. Use the Windows API call, *PostMessage*, to send a message to a control but allow... more (see page 1303)
Sending a Message Using the Windows Message Queue (see page 1303)	In a multithreaded application, you can't just call the *Perform* method because the target control is in a different thread than the one that is executing. However, by using the Windows message queue, you can safely communicate with other threads. Message handling always occurs in the main VCL thread, but you can send a message using the Windows message queue from any thread in the application. A call to *SendMessage* is synchronous. That is, *SendMessage* does not return until the target control has handled the message, even if it is in another thread. Use the Windows API call, *SendMessage*,... more (see page 1303)
Sending Messages (see page 1303)	Typically, an application sends message to send notifications of state changes or to broadcast information. Your component can broadcast messages to all the controls in a form, send messages to a particular control (or to the application itself), or even send messages to itself. There are several different ways to send a Windows message. Which method you use depends on why you are sending the message. The following topics describe the different ways to send Windows messages: • Broadcasting a message to all controls in a form. (see page 1301) • Calling a control's message handler directly. (see page 1302) • Sending a message using the Windows message queue. (see page 1303)... more (see page 1303)
Changing Message Handling (see page 1304)	Before changing the message handling of your components, make sure that is what you really want to do. The VCL translates most Windows messages into events that both the component writer and the component user can handle. Rather than changing the message-handling behavior, you should probably change the event-handling behavior. To change message handling in VCL components, you override the message-handling method. You can also prevent a component from handling a message under certain circumstances by trapping the message.
Creating New Message Handlers (see page 1304)	Because the VCL provides handlers for most common messages, the time you will most likely need to create new message handlers is when you define your own messages. Working with user-defined messages has three aspects: • Defining your own messages. (see page 1306) • Declaring a new message-handling method. (see page 1305) • Sending messages. (see page 1303)
Declaring a Message Identifier (see page 1304)	A message identifier is an integer-sized constant. Windows reserves the messages below 1,024 for its own use, so when you declare your own messages you should start above that level. The constant *WM_APP* represents the starting number for user-defined messages. When defining message identifiers, you should base them on *WM_APP*. Be aware that some standard Windows controls use messages in the user-defined range. These include list boxes, combo boxes, edit boxes, and command buttons. If you derive a component from one of these and want to define a new message for it, be sure to check the Messages... more (see page 1304)
Example: User-defined Messages (see page 1304)	The following code shows two user-defined messages.

Example: Message Structure (see page 1305)	For example, here is the message record for all mouse messages, TWMMouse, which uses a variant record to define two sets of names for the same parameters.
Declaring a Message-structure Type (see page 1305)	If you want to give useful names to the parameters of your message, you need to declare a message-record type for that message. The message-record is the type of the parameter passed to the message-handling method. If you do not use the message's parameters, or if you want to use the old-style parameter notation (wParam, lParam, and so on), you can use the default message-record, TMessage.
Declaring a New Message-handling Method (see page 1305)	There are two sets of circumstances that require you to declare new message-handling methods: • Your component needs to handle a Windows message that is not already handled by the standard components. • You have defined your own message for use by your components.
Example: Message Handler (see page 1306)	Here is the declaration of a message handler for a user-defined message called *CM_CHANGECOLOR*.
Defining Your Own Messages (see page 1306)	A number of the standard components define messages for internal use. The most common reasons for defining messages are broadcasting information not covered by standard messages and notification of state changes. You can define your own messages in the VCL. Defining a message is a two-step process. The steps are: 1. Declaring a message identifier. (see page 1304) 2. Declaring a message-record type. (see page 1305)
Dispatching Messages (see page 1306)	When an application creates a window, it registers a *window procedure* with the Windows kernel. The window procedure is the routine that handles messages for the window. Traditionally, the window procedure contains a huge **case** statement with entries for each message the window has to handle. Keep in mind that "window" in this sense means just about anything on the screen: each window, each control, and so on. Every time you create a new type of window, you have to create a complete window procedure. The VCL simplifies message dispatching in several ways: • Each component inherits a complete message-dispatching system.... more (see page 1306)
Example: Overriding a Message Handler (see page 1307)	For example, to override a component's handling of the *WM_PAINT* message, you redeclare the WMPaint method:
Overriding the Handler Method (see page 1307)	To change the way a component handles a particular message, you override the message-handling method for that message. If the component does not already handle the particular message, you need to declare a new message-handling method. To override a message-handling method, you declare a new method in your component with the same message index as the method it overrides. Do *not* use the **override** directive; you must use the **message** directive and a matching message index. Note that the name of the method and the type of the single **var** parameter do not have to match the overridden method. Only... more (see page 1307)
Handling Messages and System Notifications: Overview (see page 1308)	Components often need to respond to notifications from the underlying operating system. The operating system informs the application of occurrences such as what the user does with the mouse and keyboard. Some controls also generate notifications, such as the results from user actions such as selecting an item in a list box. The component library handles most of the common notifications already. It is possible, however, that you will need to write your own code for handling such notifications. For VCL applications, notifications arrive in the form of *messages*. These messages can come from any source, including Windows, VCL... more (see page 1308)
Trapping Messages (see page 1308)	Under some circumstances, you might want your components to ignore messages. That is, you want to keep the component from dispatching the message to its handler. To trap a message, you override the virtual method WndProc. For VCL components, the WndProc method screens messages before passing them to the Dispatch method, which in turn determines which method gets to handle the message. By overriding WndProc, your component gets a chance to filter out messages before dispatching them. An override of WndProc for a control derived from TWinControl looks like this:

The WndProc Method (see page 1308)	**Note:** This information is applicable when writing VCL components only. Here is part of the WndProc method for TControl, for example:
Understanding the message-handling system (see page 1309)	All VCL classes have a built-in mechanism for handling messages, called *message-handling methods* or *message handlers*. The basic idea of message handlers is that the class receives messages of some sort and dispatches them, calling one of a set of specified methods depending on the message received. If no specific method exists for a particular message, there is a default handler. The following diagram shows the message-dispatch system:  The Visual Component Library defines a message-dispatching system that translates all Windows messages (including user-defined messages) directed to a particular class into method calls. You should never need to alter this... more (see page 1309)
Using Message Parameters (see page 1309)	Once inside a message-handling method, your component has access to all the parameters of the message structure. Because the parameter passed to the message handler is a **var** parameter, the handler can change the values of the parameters if necessary. The only parameter that changes frequently is the Result field for the message: the value returned by the SendMessage call that sends the message. Because the type of the Message parameter in the message-handling method varies with the message being handled, you should refer to the documentation on Windows messages for the names and meanings of individual parameters. If for... more (see page 1309)
What's in a Windows Message? (see page 1309)	A Windows message is a data record that contains several fields. The most important of these is an integer-size value that identifies the message. Windows defines many messages, and the *Messages* unit declares identifiers for all of them. Other useful information in a message comes in two parameter fields and a result field. One parameter contains 16 bits, the other 32 bits. You often see Windows code that refers to those values as wParam and lParam, for *word parameter* and *long parameter*. Often, each parameter will contain more than one piece of information, and you see references to... more (see page 1309)

3.2.1.7.1 Broadcasting a Message to All Controls in a Form

When your component changes global settings that affect all of the controls in a form or other container, you may want to send a message to those controls so that they can update themselves appropriately. Not every control may need to respond to the notification, but by broadcasting the message, you can inform all controls that know how to respond and allow the other controls to ignore the message.

To broadcast a message to all the controls in another control, use the Broadcast method. Before you broadcast a message, you fill out a message record (see page 1305) with the information you want to convey.

```
var
Msg: TMessage;
begin
  Msg.Msg := MY_MYCUSTOMMESSAGE;
  Msg.WParam := 0;
  Msg.LParam := Longint(Self);
  Msg.Result := 0;
TMessage Msg;
Msg.Msg = MY_MYCUSTOMMESSAGE;
Msg.WParam = 0;
Msg.LParam = (int)(this);
Msg.Result = 0;
```

Then, pass this message record to the parent of all the controls you want to notify. This can be any control in the application. For example, it can be the parent of the control you are writing:

```
Parent.Broadcast(Msg);
Parent->Broadcast(Msg);
```

It can be the form that contains your control:

```
GetParentForm(self).Broadcast(Msg);
GetParentForm(this)->Broadcast(Msg);
```

It can be the active form:

```
Screen.ActiveForm.Broadcast(Msg);
Screen->ActiveForm->Broadcast(Msg);
```

It can even be all the forms in your application:

```
for I:= 0 to Screen.FormCount - 1 do
  Screen.Forms[I].Broadcast(Msg);
for (int i = 0; i < Screen->FormCount; i++)
  Screen->Forms[i]->Broadcast(Msg);
```

See Also

Sending a Message That Does Not Execute Immediately (see page 1303)

Sending a Message Using the Windows Message Queue (see page 1303)

3.2.1.7.2 Calling a Control's Message Handler Directly

Sometimes there is only a single control that needs to respond to your message. If you know the control that should receive your message, the simplest and most straightforward way to send the message is to call the control's Perform method.

There are two main reasons why you call a control's *Perform* method:

- You want to trigger the same response that the control makes to a standard Windows (or other) message. For example, when a grid control receives a keystroke message, it creates an inline edit control and then sends the keystroke message on to the edit control.
- You may know what control you want to notify, but not know what type of control it is. Because you don't know the type of the target control, you cannot use any of its specialized methods, but all controls have message-handling capabilities so you can always send a message. If the control has a message handler for the message you send, it will respond appropriately. Otherwise, it will ignore the message you send and return 0.

To call the *Perform* method, you do not need to create a message record. You need only pass the message identifier, WParam, and LParam as parameters. Perform returns the message result.

See Also

Sending a Message That Does Not Execute Immediately (see page 1303)

Broadcasting a Message to All Controls in a Form (see page 1301)

Sending a Message Using the Windows Message Queue (see page 1303)

3.2.1.7.3 Responding to System Events

When the widget layer receives an event notification from the operating system, it generates a special event object (*QEvent* or one of its descendants) to represent the event. The event object contains read-only information about the event that occurred. The type of the event object indicates the type of event that occurred.

The widget layer notifies your CLX component of system events using a special signal of type event. It passes the *QEvent* object to the signal handler for the event. The processing of the event signal is a bit more complicated than processing other signals because it goes first to the application object. This means an application has two opportunities to respond to a system event: once at the application level (*TApplication*) and once at the level of the individual component (your *TWidgetControl* or *THandleComponent* descendant.) All of these classes (*TApplication*, *TWidgetControl*, and *THandleComponent*) already assign a signal handler for the event signal from the widget layer. That is, all system events are automatically directed to the *EventFilter* method, which plays a role similar to the *WndProc* method on VCL controls.

EventFilter handles most of the commonly used system notifications, translating them into the events that are introduced by your component's base classes. Thus, for example, the *EventFilter* method of *TWidgetControl* responds to mouse events (*QMouseEvent*) by generating the *OnMouseDown*, *OnMouseMove*, and *OnMouseUp* events, to keyboard events (*QKeyEvent*)

by generating the *OnKeyDown*, *OnKeyPress*, *OnKeyString*, and *OnKeyUp* events, and so on.

The following topics describe how to customize the way your control works with system events:

- Commonly used events
- Overriding the EventFilter method
- Generating Qt events

3.2.1.7.4 Sending a Message That Does Not Execute Immediately

There are times you may want to send a message but you do not know whether it is safe for the target of the message to execute right away. For example, if the code that sends a message is called from an event handler on the target control, you may want to make sure that the event handler has finished executing before the control executes your message. You can handle this situation as long as you do not need to know the message result.

Use the Windows API call, *PostMessage*, to send a message to a control but allow the control to wait until it has finished any other messages before it handles yours. *PostMessage* takes exactly the same parameters as *SendMessage*.

For more information on the PostMessage function, see the Microsoft MSDN documentation.

See Also

Broadcasting a Message to All Controls in a Form (see page 1301)

Sending a Message Using the Windows Message Queue (see page 1303)

3.2.1.7.5 Sending a Message Using the Windows Message Queue

In a multithreaded application, you can't just call the *Perform* method because the target control is in a different thread than the one that is executing. However, by using the Windows message queue, you can safely communicate with other threads. Message handling always occurs in the main VCL thread, but you can send a message using the Windows message queue from any thread in the application. A call to *SendMessage* is synchronous. That is, *SendMessage* does not return until the target control has handled the message, even if it is in another thread.

Use the Windows API call, *SendMessage*, to send a message to a control using the Windows message queue. *SendMessage* takes the same parameters as the *Perform* method, except that you must identify the target control by passing its Window handle. Thus, instead of writing

```
MsgResult := TargetControl.Perform(MY_MYMESSAGE, 0, 0);
MsgResult = TargetControl->Perform(MY_MYMESSAGE, 0, 0);
```

you would write

```
MsgResult := SendMessage(TargetControl.Handle, MYMESSAGE, 0, 0);
MsgResult = SendMessage(TargetControl->Handle, MYMESSAGE, 0, 0);
```

For more information on the SendMessage function, see the Microsoft MSDN documentation. For more information on writing multiple threads that may be executing simultaneously, see Coordinating threads (see page 2227).

See Also

Sending a Message That Does Not Execute Immediately (see page 1303)

Broadcasting a Message to All Controls in a Form (see page 1301)

3.2.1.7.6 Sending Messages

Typically, an application sends message to send notifications of state changes or to broadcast information. Your component can

broadcast messages to all the controls in a form, send messages to a particular control (or to the application itself), or even send messages to itself.

There are several different ways to send a Windows message. Which method you use depends on why you are sending the message. The following topics describe the different ways to send Windows messages:

- Broadcasting a message to all controls in a form. (see page 1301)
- Calling a control's message handler directly. (see page 1302)
- Sending a message using the Windows message queue. (see page 1303)
- Sending a message that does not execute immediately. (see page 1303)

3.2.1.7.7 Changing Message Handling

Before changing the message handling of your components, make sure that is what you really want to do. The VCL translates most Windows messages into events that both the component writer and the component user can handle. Rather than changing the message-handling behavior, you should probably change the event-handling behavior.

To change message handling in VCL components, you override the message-handling method. You can also prevent a component from handling a message under certain circumstances by trapping the message.

3.2.1.7.8 Creating New Message Handlers

Because the VCL provides handlers for most common messages, the time you will most likely need to create new message handlers is when you define your own messages. Working with user-defined messages has three aspects:

- Defining your own messages. (see page 1306)
- Declaring a new message-handling method. (see page 1305)
- Sending messages. (see page 1303)

3.2.1.7.9 Declaring a Message Identifier

A message identifier is an integer-sized constant. Windows reserves the messages below 1,024 for its own use, so when you declare your own messages you should start above that level.

The constant *WM_APP* represents the starting number for user-defined messages. When defining message identifiers, you should base them on *WM_APP*.

Be aware that some standard Windows controls use messages in the user-defined range. These include list boxes, combo boxes, edit boxes, and command buttons. If you derive a component from one of these and want to define a new message for it, be sure to check the Messages unit to see which messages Windows already defines for that control.

3.2.1.7.10 Example: User-defined Messages

The following code shows two user-defined messages.

```
const
  MY_MYFIRSTMESSAGE = WM_APP + 400;
  MY_MYSECONDMESSAGE = WM_APP + 401;
#define MY_MYFIRSTMESSAGE  (WM_APP + 400)
#define MY_MYSECONDMESSAGE (WM_APP + 401)
```

3.2.1.7.11 Example: Message Structure

For example, here is the message record for all mouse messages, TWMMouse, which uses a variant record to define two sets of names for the same parameters.

```
type
  TWMMouse = record
    Msg: TMsgParam;        ( first is the message ID )
    Keys: Word;            ( this is the wParam )
    case Integer of        ( two ways to look at the lParam )
      0: {
         XPos: Integer;    ( either as x- and y-coordinates...)
         YPos: Integer);
      1: {
         Pos: TPoint;      ( ... or as a single point )
         Result: Longint); ( and finally, the result field )
end;
struct TWMKey
{
  Cardinal Msg;                    // first parameter is the message ID
  Word CharCode;                   // this is the first wParam
  Word Unused;
  Longint KeyData;                 // this is the lParam
  Longint Result;                  // this is the result data member
};
```

3.2.1.7.12 Declaring a Message-structure Type

If you want to give useful names to the parameters of your message, you need to declare a message-record type for that message. The message-record is the type of the parameter passed to the message-handling method. If you do not use the message's parameters, or if you want to use the old-style parameter notation (wParam, lParam, and so on), you can use the default message-record, TMessage.

To declare a message-record type, follow these conventions:

1. Name the record type after the message, preceded by a *T*.
2. Call the first field in the record `Msg`, of type TMsgParam.
3. Define the next two bytes to correspond to the Word parameter, and the next two bytes as unused. Or Define the next four bytes to correspond to the Longint parameter.
4. Add a final field called Result, of type Longint.

3.2.1.7.13 Declaring a New Message-handling Method

There are two sets of circumstances that require you to declare new message-handling methods:

- Your component needs to handle a Windows message that is not already handled by the standard components.
- You have defined your own message for use by your components.

To declare a message-handling method, do the following:

1. Declare the method in a **protected** part of the component's class declaration.
2. Make the method a procedure.
3. Name the method after the message it handles, but without any underline characters.
4. Pass a single **var** parameter called Message, of the type of the message record.
5. Within the message method implementation, write code for any handling specific to the component.

6. Call the inherited message handler.

3.2.1.7.14 Example: Message Handler

Here is the declaration of a message handler for a user-defined message called *CM_CHANGECOLOR*.

```
const
  CM_CHANGECOLOR = WM_APP + 400;
type
  TMyComponent = class(TControl)
  .
  .
  .
protected
  procedure CMChangeColor(var Message: TMessage); message CM_CHANGECOLOR;
end;
procedure TMyComponent.CMChangeColor(var Message: TMessage);
begin
  Color := Message.lParam;
  inherited;
end;
#define CM_CHANGECOLOR (WM_APP + 400)
class TMyControl : public TControl
{
protected:
    void __fastcall CMChangeColor(TMessage &Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_CHANGECOLOR, TMessage, CMChangeColor)
END_MESSAGE_MAP(TControl)
};
void __fastcall TMyControl::CMChangeColor(TMessage &Message)
{
  Color = Message.LParam;                    // set color from long parameter
  TControl::CMChangeColor(Message);          // call the inherited message handler
}
```

3.2.1.7.15 Defining Your Own Messages

A number of the standard components define messages for internal use. The most common reasons for defining messages are broadcasting information not covered by standard messages and notification of state changes. You can define your own messages in the VCL.

Defining a message is a two-step process. The steps are:

1. Declaring a message identifier. (see page 1304)
2. Declaring a message-record type. (see page 1305)

3.2.1.7.16 Dispatching Messages

When an application creates a window, it registers a *window procedure* with the Windows kernel. The window procedure is the routine that handles messages for the window. Traditionally, the window procedure contains a huge **case** statement with entries for each message the window has to handle. Keep in mind that "window" in this sense means just about anything on the screen: each window, each control, and so on. Every time you create a new type of window, you have to create a complete window procedure.

The VCL simplifies message dispatching in several ways:

- Each component inherits a complete message-dispatching system.
- The dispatch system has default handling. You define handlers only for messages you need to respond to specially.

- You can modify small parts of the message handling and rely on inherited methods for most processing.

The greatest benefit of this message dispatch system is that you can safely send any message to any component at any time. If the component does not have a handler defined for the message, the default handling takes care of it, usually by ignoring the message.

Tracing the flow of messages

The VCL registers a method called MainWndProc as the window procedure for each type of component in an application. MainWndProc contains an exception-handling block, passing the message structure from Windows to a virtual method called WndProc and handling any exceptions by calling the application class's HandleException method.

MainWndProc is a nonvirtual method that contains no special handling for any particular messages. Customizations take place in WndProc, since each component type can override the method to suit its particular needs.

WndProc methods check for any special conditions that affect their processing so they can "trap" unwanted messages. For example, while being dragged, components ignore keyboard events, so the WndProc method of TWinControl passes along keyboard events only if the component is not being dragged. Ultimately, WndProc calls Dispatch, a nonvirtual method inherited from TObject, which determines which method to call to handle the message.

Dispatch uses the Msg field of the message structure to determine how to dispatch a particular message. If the component defines a handler for that particular message, Dispatch calls the method. If the component does not define a handler for that message, Dispatch calls DefaultHandler.

3.2.1.7.17 Example: Overriding a Message Handler

For example, to override a component's handling of the *WM_PAINT* message, you redeclare the WMPaint method:

```
type
  TMyComponent = class(...)
  .
  .
  .
  procedure WMPaint(var Message: TWMPaint); message WM_PAINT;
end;
```

```
class PACKAGE TMyComponent : public TComponent
{
protected:
    void __fastcall WMPaint(TWMPaint* Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(WM_PAINT, TWMPaint, WMPaint)
END_MESSAGE_MAP(TComponent)
};
```

3.2.1.7.18 Overriding the Handler Method

To change the way a component handles a particular message, you override the message-handling method for that message. If the component does not already handle the particular message, you need to declare a new message-handling method.

To override a message-handling method, you declare a new method in your component with the same message index as the method it overrides. Do *not* use the **override** directive; you must use the **message** directive and a matching message index.

Note that the name of the method and the type of the single **var** parameter do not have to match the overridden method. Only the message index is significant. For clarity, however, it is best to follow the convention of naming message-handling methods after the messages they handle.

```
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(parameter1, parameter2, parameter3)
END_MESSAGE_MAP
```

3.2.1.7.19 Handling Messages and System Notifications: Overview

Components often need to respond to notifications from the underlying operating system. The operating system informs the application of occurrences such as what the user does with the mouse and keyboard. Some controls also generate notifications, such as the results from user actions such as selecting an item in a list box. The component library handles most of the common notifications already. It is possible, however, that you will need to write your own code for handling such notifications.

For VCL applications, notifications arrive in the form of *messages*. These messages can come from any source, including Windows, VCL components, and components you have defined. There are three aspects to working with messages:

- Understanding the message-handling system. (see page 1309)
- Changing message handling. (see page 1304)
- Creating new message handlers. (see page 1304)

3.2.1.7.20 Trapping Messages

Under some circumstances, you might want your components to ignore messages. That is, you want to keep the component from dispatching the message to its handler. To trap a message, you override the virtual method WndProc.

For VCL components, the WndProc method screens messages before passing them to the Dispatch method, which in turn determines which method gets to handle the message. By overriding WndProc, your component gets a chance to filter out messages before dispatching them. An override of WndProc for a control derived from TWinControl looks like this:

```
procedure TMyControl.WndProc(var Message: TMessage);
begin
  { tests to determine whether to continue processing }
  inherited WndProc(Message);
end;
void __fastcall TMyControl::WndProc(TMessage& Message)
{
   // tests to determine whether to continue processing
if(Message.Msg != WM_LBUTTONDOWN)

}
```

The TControl component defines entire ranges of mouse messages that it filters when a user is dragging and dropping controls. Overriding WndProc helps this in two ways:

- It can filter ranges of messages instead of having to specify handlers for each one.
- It can preclude dispatching the message at all, so the handlers are never called.

3.2.1.7.21 The WndProc Method

Note: This information is applicable when writing VCL components only.

Here is part of the WndProc method for TControl, for example:

```
procedure TControl.WndProc(var Message: TMessage);
begin
  .
  .
  .
  if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST) then
    if Dragging then                                    { handle dragging specially }
      DragMouseMsg(TWMMouse(Message))
    else
      .                                                 { handle others normally }
      .
```

```
      .
    end;
.                                                    { otherwise process normally }
.
.
end;
```

3.2.1.7.22 Understanding the message-handling system

All VCL classes have a built-in mechanism for handling messages, called *message-handling methods* or *message handlers*. The basic idea of message handlers is that the class receives messages of some sort and dispatches them, calling one of a set of specified methods depending on the message received. If no specific method exists for a particular message, there is a default handler.

The following diagram shows the message-dispatch system:



The Visual Component Library defines a message-dispatching system that translates all Windows messages (including user-defined messages) directed to a particular class into method calls. You should never need to alter this message-dispatch mechanism. All you will need to do is create message-handling methods. See the section Declaring a new message-handling method (see page 1305) for more on this subject.

3.2.1.7.23 Using Message Parameters

Once inside a message-handling method, your component has access to all the parameters of the message structure. Because the parameter passed to the message handler is a **var** parameter, the handler can change the values of the parameters if necessary. The only parameter that changes frequently is the Result field for the message: the value returned by the SendMessage call that sends the message.

Because the type of the Message parameter in the message-handling method varies with the message being handled, you should refer to the documentation on Windows messages for the names and meanings of individual parameters. If for some reason you need to refer to the message parameters by their old-style names (WParam, LParam, and so on), you can typecast Message to the generic type TMessage, which uses those parameter names.

3.2.1.7.24 What's in a Windows Message?

A Windows message is a data record that contains several fields. The most important of these is an integer-size value that identifies the message. Windows defines many messages, and the *Messages* unit declares identifiers for all of them. Other useful information in a message comes in two parameter fields and a result field.

One parameter contains 16 bits, the other 32 bits. You often see Windows code that refers to those values as wParam and lParam, for *word parameter* and *long parameter*. Often, each parameter will contain more than one piece of information, and you see references to names such as lParamHi, which refers to the high-order word in the long parameter.

Originally, Windows programmers had to remember or look up in the Windows APIs what each parameter contained. Now Microsoft has named the parameters. This so-called *message cracking* makes it much simpler to understand what information accompanies each message. For example, the parameters to the *WM_KEYDOWN* message are now called nVirtKey and lKeyData, which gives much more specific information than wParam and lParam.

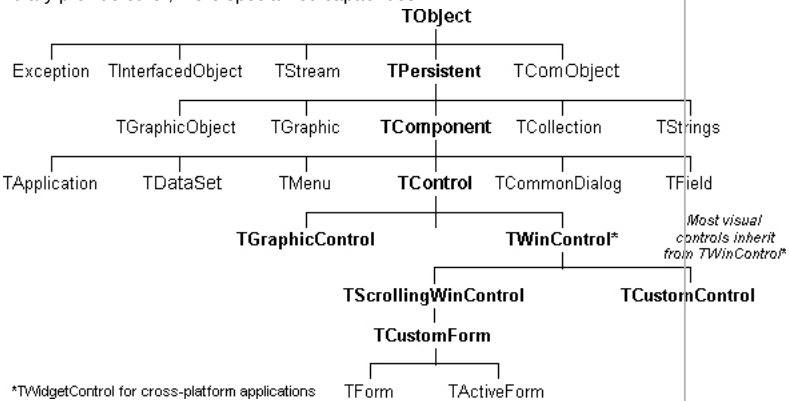
For each type of message, Delphi defines a record type that gives a mnemonic name to each parameter. For example, mouse messages pass the x- and y-coordinates of the mouse event in the long parameter, one in the high-order word, and the other in the low-order word. Using the mouse-message structure, you do not have to worry about which word is which, because you refer to the parameters by the names XPos and YPos instead of lParamLo and lParamHi.

```
void MyKeyDownHandler(  HWND hwnd, UINT nVirtKey, BOOL fDown, int CRepeat, UINT flags )
```

```
{
   .
   .
   .
}
LRESULT MyWndProc( HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam )
{
  switch(  Message )
  {
    HANDLE_MSG( hwnd, WM_KEYDOWN, MyKeyDownHandler );
     .
     .
     .
}
```

3.2.1.8 Introduction to component creation

Topics

Name	Description
Overview of Component Creation (see page 1313)	This set of topics provides an overview of component design and the process of writing components for Delphi applications. The material here assumes that you are familiar with Delphi and its standard components. The main topics discussed are • Class library (see page 1314) • Components and classes (see page 1315) • Creating components (see page 1315) • What goes into a component? (see page 1323) • Creating a new component (see page 1317) • Testing uninstalled components (see page 1324) • Testing installed components (see page 1324) For information on installing new components, see Installing component packages (see page 2217).
Class library (see page 1314)	Delphi's components reside in the Visual Component Library (VCL) . The following figure shows the relationship of selected classes that make up the VCL hierarchy. For a more detailed discussion of class hierarchies and the inheritance relationships among classes, see Object-oriented programming for component writers (see page 1369) The *TComponent* class is the shared ancestor of every component in the component library. *TComponent* provides the minimal properties and events necessary for a component to work in the IDE. The various branches of the library provide other, more specialized capabilities.  When you create a component, you add to the component library by deriving a... more (see page 1314)

Components and Classes (see page 1315)	Because components are classes, component writers work with objects at a different level from application developers. Creating new components requires that you derive new classes. Briefly, there are two main differences between creating components and using them in applications. When creating components, • You access parts of the class that are inaccessible to application programmers. • You add new parts (such as properties) to your components. Because of these differences, you need to be aware of more conventions and think about how application developers will use the components you write.
Creating Components (see page 1315)	A component can be almost any program element that you want to manipulate at design time. Creating a component means deriving a new class from an existing one. You can derive a new component in several ways: • Modifying existing controls (see page 1316) • Creating windowed controls (see page 1316) • Creating graphic controls (see page 1318) • Subclassing Windows controls (see page 1321) • Creating nonvisual components (see page 1318) The following table summarizes the different kinds of components and the classes you use as starting points for each. ***Component creation starting points***
Modifying Existing Controls (see page 1316)	The simplest way to create a component is to customize an existing one. You can derive a new component from any of the components provided in the component library. Some controls, such as list boxes and grids, come in several variations on a basic theme. In these cases, the component library includes an abstract class (with the word "custom" in its name, such as *TCustomGrid*) from which to derive customized versions. For example, you might want to create a special list box that does not have some of the properties of the standard *TListBox* class. You cannot remove (hide)... more (see page 1316)
Creating Original Controls (see page 1316)	Windowed controls in the component library are objects that appear at runtime and that the user can interact with. Each windowed control has a window handle, accessed through its *Handle* property, that lets the operating system identify and operate on the control. If using VCL controls, the handle allows the control to receive input focus and can be passed to Windows API functions. Each widget-based control has a handle, accessed through its *Handle* property, that identifies the underlying widget.--> All windowed controls descend from the *TWinControl*class. These include most standard windowed controls, such as pushbuttons, list boxes, and edit... more (see page 1316)
Creating a New Component (see page 1317)	This topic describes how to create and setup a component.
Creating a Unit File (see page 1317)	A unit is a separately compiled module of Delphi code. Delphi uses units for several purposes. Every form has its own unit, and most components (or groups of related components) have their own units as well. When you create a component, you either create a new unit for the component or add the new component to an existing unit.
Declaring A New Constructor (C++) (see page 1318)	Each new component must have a constructor that overrides the constructor of the class from which it was derived. When you write the constructor for your new component, it must always call the inherited constructor. Within the class declaration, declare a virtual constructor in the public section of the class. For example,

Deriving the Component (see page 1318)	Every component is a class derived from *TComponent*, from one of its more specialized descendants (such as *TControl* or *TGraphicControl*), or from an existing component class. The section Creating components (see page 1315) describes which class to derive different kinds of components from. Deriving classes is explained in more detail in The section Defining new classes. (see page 1370) To derive a component, add an object type declaration to the **interface** part of the unit that will contain the component. A simple component class is a nonvisual component descended directly from *TComponent*.
Creating Nonvisual Components (see page 1318)	Nonvisual components are used as interfaces for elements like databases (*TDataSet or TSQLConnection*) and system clocks (*TTimer*), and as placeholders for dialog boxes (*TCommonDialog* and its descendants). Most of the components you write are likely to be visual controls. Nonvisual components can be derived directly from *TComponent*, the abstract base class for all components.
Creating Graphic Controls (see page 1318)	If your control does not need to receive input focus, you can make it a graphic control. Graphic controls are similar to windowed controls, but have no window handles, and therefore consume fewer system resources. Components like *TLabel*, which never receive input focus, are graphic controls. Although these controls cannot receive focus, you can design them to react to mouse messages. You can create custom controls through the *TGraphicControl* component. *TGraphicControl* is an abstract class derived from *TControl*. Although you can derive controls directly from *TControl*, it is better to start from *TGraphicControl*, which provides a... more (see page 1318)
Encapsulating Graphics (see page 1319)	Delphi simplifies Windows graphics by encapsulating various graphics tools into a canvas. The canvas represents the drawing surface of a window or control and contains other classes, such as a pen, a brush, and a font. A canvas is like a Windows device context, but it takes care of all the bookkeeping for you. If you have written a graphical Windows application, you are familiar with the requirements imposed by Windows' graphics device interface (GDI). For example, GDI limits the number of device contexts available and requires that you restore graphic objects to their initial state before destroying them. With... more (see page 1319)
Setting Properties, Methods, and Events (see page 1319)	Aside from the visible image manipulated in the Form designer, the most obvious attributes of a component are its properties, events, and methods. Each of these has a section devoted to it in this file, but the discussion that follows explains some of the motivation for their use.
Registering the Component (see page 1320)	Registration is a simple process that tells the IDE which components to add to its component library, and on which pages of the Tool palette they should appear. For a more detailed discussion of the registration process, see Making components available at design time (see page 1352)
Removing Dependencies (see page 1321)	One quality that makes components usable is the absence of restrictions on what they can do at any point in their code. By their nature, components are incorporated into applications in varying combinations, orders, and contexts. You should design components that function in any situation, without preconditions. An example of removing dependencies is the *Handle* property of *TWinControl*. If you have written Windows applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you do not try to access a windowed control until you have created it... more (see page 1321)
Registering Components (Introduction) (see page 1321)	Before you can install your components in the IDE, you have to register them. Registration tells Delphi where to place the component on the Tool palette. You can also customize the way Delphi stores your components in the form file. For information on registering a component, see Registering components. (see page 1352)
Subclassing Windows Controls (see page 1321)	In traditional Windows programming, you create custom controls by defining a new *window class* and registering it with Windows. The window class (which is similar to the *objects* or *classes* in object-oriented programming) contains information shared among instances of the same sort of control; you can base a new window class on an existing class, which is called *subclassing*. You then put your control in a dynamic-link library (DLL), much like the standard Windows controls, and provide an interface to it. You can create a component "wrapper" around any existing window class. So if you already have a library... more (see page 1321)

Creating a Component with the Component Wizard (see page 1321)	The Component wizard simplifies the initial stages of creating a component. When you use the Component wizard, you need to specify: • The class from which the component is derived. • The class name for the new component. • The Tool palette category where you want it to appear. • The name of the unit in which the component is created. • The search path where the unit is found. • The name of the package in which you want to place the component. The Component wizard performs the same tasks you would when creating a component manually: • Creating a unit. • Deriving the component. • Registering the... more (see page 1321)
What Goes into a Component? (see page 1323)	To make your components reliable parts of the Delphi environment, you need to follow certain conventions in their design. This section discusses the following topics: • Removing dependencies (see page 1321) • Setting properties (see page 1319) • Encapsulating graphics (see page 1319) • Registering components (see page 1321)
Making Source Files Available (see page 1323)	Component writers should make all source files used by a component should be located in the same directory. These files include source code files (.pas) and additional project files (.dfm/.xfm, .res, .rc, and .dcr). The process of adding a component results in the creation of a number of files. These files are automatically put in directories specified in the IDE environment options (use the menu command **Tools** ▶ **Options**, navigate to the **Environment Options** ▶ **Delphi Options** ▶ **Library** page). The .lib files are placed in the DCP output directory. If adding the component entails creating a new package (as opposed to... more (see page 1323)
Testing Installed Components (see page 1324)	You can test the design-time behavior of a component after you install it on the Tool palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Tool palette. For information on testing components that have not yet been installed, see Testing uninstalled components (see page 1324). Testing your components after installing allows you to debug the component that only generates design-time exceptions when dropped on a form.
Testing Uninstalled Components (see page 1324)	You can test the runtime behavior of a component before you install it on the Tool palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Tool palette. For information on testing already installed components, see Testing installed components (see page 1324). You test an uninstalled component by emulating the actions performed by Delphi when the component is selected from the palette and placed on a form.

3.2.1.8.1 Overview of Component Creation

This set of topics provides an overview of component design and the process of writing components for Delphi applications. The material here assumes that you are familiar with Delphi and its standard components.

The main topics discussed are

- Class library (see page 1314)
- Components and classes (see page 1315)

- Creating components (see page 1315)
- What goes into a component? (see page 1323)
- Creating a new component (see page 1317)
- Testing uninstalled components (see page 1324)
- Testing installed components (see page 1324)

For information on installing new components, see Installing component packages (see page 2217).

See Also

Object-oriented Programming for Component Writers: Overview (see page 1369)

Creating Properties: Overview (see page 1249)

Creating Events: Overview (see page 1233)

Creating Methods: Overview (see page 1243)

Using Graphics in Components: Overview (see page 1380)

Responding to Messages (see page 1308)

Making Components Available at Design Time: Overview (see page 1350)

Working with Frames (see page 2012)

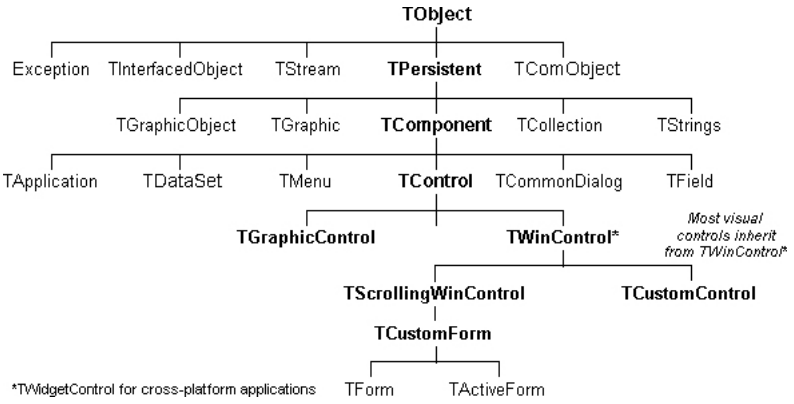
Creating and Using Component Templates (see page 1975)

Reusing Components and Groups of Components (see page 2000)

3.2.1.8.2 Class library

Delphi's components reside in the Visual Component Library (VCL) . The following figure shows the relationship of selected classes that make up the VCL hierarchy. For a more detailed discussion of class hierarchies and the inheritance relationships among classes, see Object-oriented programming for component writers (see page 1369)

The *TComponent* class is the shared ancestor of every component in the component library. *TComponent* provides the minimal properties and events necessary for a component to work in the IDE. The various branches of the library provide other, more specialized capabilities.



When you create a component, you add to the component library by deriving a new class from one of the existing class types in the hierarchy.

See Also

Components and Objects (see page 1315)

What Goes into a Component (see page 1323)

Creating a New Component (see page 1317)

Testing Uninstalled Components (see page 1324)

3.2.1.8.3 Components and Classes

Because components are classes, component writers work with objects at a different level from application developers. Creating new components requires that you derive new classes.

Briefly, there are two main differences between creating components and using them in applications. When creating components,

- You access parts of the class that are inaccessible to application programmers.
- You add new parts (such as properties) to your components.

Because of these differences, you need to be aware of more conventions and think about how application developers will use the components you write.

See Also

Components and Classes

Creating Components (see page 1315)

What Goes into a Component (see page 1323)

Creating a New Component (see page 1317)

Testing Uninstalled Components (see page 1324)

3.2.1.8.4 Creating Components

A component can be almost any program element that you want to manipulate at design time. Creating a component means deriving a new class from an existing one. You can derive a new component in several ways:

- Modifying existing controls (see page 1316)
- Creating windowed controls (see page 1316)
- Creating graphic controls (see page 1318)
- Subclassing Windows controls (see page 1321)
- Creating nonvisual components (see page 1318)

The following table summarizes the different kinds of components and the classes you use as starting points for each.

Component creation starting points

To do this	Start with this type
Modify an existing component	Any existing component, such as *TButton* or *TListBox*, or an abstract component type, such as *TCustomListBox*
Create a windowed control	*TWinControl*
Create a graphic control	*TGraphicControl*
Subclassing a control	Any Windows control
Create a nonvisual component	*TComponent*

You can also derive classes that are not components and cannot be manipulated on a form, such as *TRegIniFile* and *TFont*.

See Also

The Visual Component Library (see page 1314)

Components and Classes (see page 1315)

What Goes into a Component (see page 1323)

Creating a New Component (see page 1317)

Testing Uninstalled Components (see page 1324)

3.2.1.8.5 Modifying Existing Controls

The simplest way to create a component is to customize an existing one. You can derive a new component from any of the components provided in the component library.

Some controls, such as list boxes and grids, come in several variations on a basic theme. In these cases, the component library includes an abstract class (with the word "custom" in its name, such as *TCustomGrid*) from which to derive customized versions.

For example, you might want to create a special list box that does not have some of the properties of the standard *TListBox* class. You cannot remove (hide) a property inherited from an ancestor class, so you need to derive your component from something above *TListBox* in the hierarchy. Rather than force you to start from the abstract *TWinControl*class and reinvent all the list box functions, the component library provides *TCustomListBox*, which implements the properties of a list box but does not publish all of them. When you derive a component from an abstract class like *TCustomListBox*, you publish only the properties you want to make available in your component and leave the rest protected.

The section Creating properties (see page 1249) explains publishing inherited properties. The section Modifying an existing component (see page 1365) and the section Customizing a grid (see page 1264) show examples of modifying existing controls.

See Also

Generic Components (see page 1318)

Windowed Controls (see page 1316)

Graphical Controls (see page 1318)

Subclassed Controls (see page 1321)

3.2.1.8.6 Creating Original Controls

Windowed controls in the component library are objects that appear at runtime and that the user can interact with. Each windowed control has a window handle, accessed through its *Handle* property, that lets the operating system identify and operate on the control. If using VCL controls, the handle allows the control to receive input focus and can be passed to Windows API functions. Each widget-based control has a handle, accessed through its *Handle* property, that identifies the underlying widget.-->

All windowed controls descend from the *TWinControl*class. These include most standard windowed controls, such as pushbuttons, list boxes, and edit boxes. While you could derive an original control (one that's not related to any existing control) directly from *TWinControl*, Delphi provides the *TCustomControl* component for this purpose. *TCustomControl* is a specialized windowed control that makes it easier to draw complex visual images.

The section Customizing a grid (see page 1264) presents an example of creating a windowed control.

3.2.1.8.7 Creating a New Component

This topic describes how to create and setup a component.

To create a component, follow these steps:

1. Creating a unit file (see page 1317)
2. Deriving the component (see page 1318)
3. Registering the component (see page 1320)

Now you will have a minimally functional component ready to install on the Tool palette. After installing, you can add your new component to a form and test it at both design time and runtime. You can then add more features to the component, update the Tool palette, and continue testing.

There are several basic steps that you perform whenever you create a new component. These steps are described below; other examples in this document assume that you know how to perform them.

1. Create a unit for the new component.
2. Derive your component from an existing component type.
3. Add properties, methods, and events.
4. Register your component with the IDE.
5. Create a bitmap for the component.
6. Create a package (a special dynamic-link library) so that you can install your component in the IDE.
7. Create a Help file for your component and its properties, methods, and events.

   Note: Creating a Help file to instruct component users on how to use the component is optional.

   When you finish, the complete component includes the following files:

- A package (.BPL) or package collection (.DPC) file
- A compiled package (.DCP) file
- A compiled unit (.DCU) file
- A palette bitmap (.DCR) file
- A Help (.HLP) file

You can also create a bitmap to represent your new component. .

3.2.1.8.8 Creating a Unit File

A unit is a separately compiled module of Delphi code. Delphi uses units for several purposes. Every form has its own unit, and most components (or groups of related components) have their own units as well.

When you create a component, you either create a new unit for the component or add the new component to an existing unit.

To create a new unit for a component:

1. Choose either:

- **File ▶ New ▶ Unit**.
- **File ▶ New ▶ Other** to display the New Items dialog box, select **Delphi Projects ▶ Delphi Files ▶ Unit**, and choose OK. The IDE creates a new unit file and opens it in the Code editor.

2. Save the file with a meaningful name.
3. Derive the component class.

To open an existing unit:

1. Choose **File** ▸ **Open** and select the source code unit to which you want to add your component.

   Note: When adding a component to an existing unit, make sure that the unit contains only component code. For example, adding component code to a unit that contains a form causes errors in the Tool palette.

2. Derive the component class.

3.2.1.8.9 Declaring A New Constructor (C++)

Each new component must have a constructor that overrides the constructor of the class from which it was derived. When you write the constructor for your new component, it must always call the inherited constructor.

Within the class declaration, declare a virtual constructor in the public section of the class.

For example,

```
class PACKAGE TNewComponent : public TComponent
{
public:
  virtual __fastcall TNewComponent(TComponent* AOwner);
};
```

In the .CPP file, implement the constructor:

```
__fastcall TNewComponent::TNewComponent(TComponent* AOwner): TComponent(AOwner)
{
}
```

Within the constructor, you add the code you want to execute when the component is created.

3.2.1.8.10 Deriving the Component

Every component is a class derived from *TComponent*, from one of its more specialized descendants (such as *TControl* or *TGraphicControl*), or from an existing component class. The section Creating components (see page 1315) describes which class to derive different kinds of components from.

Deriving classes is explained in more detail in The section Defining new classes. (see page 1370)

To derive a component, add an object type declaration to the **interface** part of the unit that will contain the component.

A simple component class is a nonvisual component descended directly from *TComponent*.

3.2.1.8.11 Creating Nonvisual Components

Nonvisual components are used as interfaces for elements like databases (*TDataSet or TSQLConnection*) and system clocks (*TTimer*), and as placeholders for dialog boxes (*TCommonDialog* and its descendants). Most of the components you write are likely to be visual controls. Nonvisual components can be derived directly from *TComponent*, the abstract base class for all components.

3.2.1.8.12 Creating Graphic Controls

If your control does not need to receive input focus, you can make it a graphic control. Graphic controls are similar to windowed controls, but have no window handles, and therefore consume fewer system resources. Components like *TLabel*, which never receive input focus, are graphic controls. Although these controls cannot receive focus, you can design them to react to mouse

messages.

You can create custom controls through the *TGraphicControl* component. *TGraphicControl* is an abstract class derived from *TControl*. Although you can derive controls directly from *TControl*, it is better to start from *TGraphicControl*, which provides a canvas to paint on and on Windows, handles *WM_PAINT* messages; all you need to do is override the *Paint* method.

The section Creating a graphic control (see page 1220) presents an example of creating a graphic control.

3.2.1.8.13 Encapsulating Graphics

Delphi simplifies Windows graphics by encapsulating various graphics tools into a canvas. The canvas represents the drawing surface of a window or control and contains other classes, such as a pen, a brush, and a font. A canvas is like a Windows device context, but it takes care of all the bookkeeping for you.

If you have written a graphical Windows application, you are familiar with the requirements imposed by Windows' graphics device interface (GDI). For example, GDI limits the number of device contexts available and requires that you restore graphic objects to their initial state before destroying them.

With Delphi, you do not have to worry about these things. To draw on a form or other component, you access the component's *Canvas* property. If you want to customize a pen or brush, you set its color or style. When you finish, Delphi disposes of the resources. Delphi caches resources to avoid recreating them if your application frequently uses the same kinds of resource.

You still have full access to the Windows GDI, but you will often find that your code is simpler and runs faster if you use the canvas built into Delphi components.

How graphics images work in the component depends on the canvas of the object from which your component descends. Graphics features are detailed in the section Using graphics in components. (see page 1380)

3.2.1.8.14 Setting Properties, Methods, and Events

Aside from the visible image manipulated in the Form designer, the most obvious attributes of a component are its properties, events, and methods. Each of these has a section devoted to it in this file, but the discussion that follows explains some of the motivation for their use.

Properties

Properties give the application developer the illusion of setting or reading the value of a variable, while allowing the component writer to hide the underlying data structure or to implement special processing when the value is accessed.

There are several advantages to using properties:

- Properties are available at design time. The application developer can set or change initial values of properties without having to write code.
- Properties can check values or formats as the application developer assigns them. Validating input at design time prevents errors.
- The component can construct appropriate values on demand. Perhaps the most common type of error programmers make is to reference a variable that has not been initialized. By representing data with a property, you can ensure that a value is always available on demand.
- Properties allow you to hide data under a simple, consistent interface. You can alter the way information is structured in a property without making the change visible to application developers.

The section Overview of component creation (see page 1249) explains how to add properties to your components.

Methods

Class methods are procedures and functions that operate on a class rather than on specific instances of the class. For example, every component's constructor method (*Create*) is a class method. Component methods are procedures and functions that

operate on the component instances themselves. Application developers use methods to direct a component to perform a specific action or return a value not contained by any property.

Because they require execution of code, methods can be called only at runtime. Methods are useful for several reasons:

- Methods encapsulate the functionality of a component in the same object where the data resides.
- Methods can hide complicated procedures under a simple, consistent interface. An application developer can call a component's *AlignControls* method without knowing how the method works or how it differs from the *AlignControls* method in another component.
- Methods allow updating of several properties with a single call.

The section Creating methods (see page 1243) explains how to add methods to your components.

Events

An event is a special property that invokes code in response to input or other activity at runtime. Events give the application developer a way to attach specific blocks of code to specific runtime occurrences, such as mouse actions and keystrokes. The code that executes when an event occurs is called an *event handler*.

Events allow application developers to specify responses to different kinds of input without defining new components.

The section Creating events (see page 1233) explains how to implement standard events and how to define new ones.

3.2.1.8.15 Registering the Component

Registration is a simple process that tells the IDE which components to add to its component library, and on which pages of the Tool palette they should appear. For a more detailed discussion of the registration process, see Making components available at design time (see page 1352)

To register a component:

1. Add a procedure named *Register* to the **interface** part of the component's unit. *Register* takes no parameters, so the declaration is very simple:

```
procedure Register;
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
  }
}
TComponentClass classes[1] = {__classid(TNewComponent)};
```

If you are adding a component to a unit that already contains components, it should already have a *Register* procedure declared, so you do not need to change the declaration.

Note: Although Delphi is a case insensitive language, the *Register* procedure is case sensitive and must be spelled with an uppercase R.

2. Write the *Register* procedure in the **implementation** part of the unit, calling *RegisterComponents* for each component you want to register. *RegisterComponents* is a procedure that takes two parameters: the name of a Tool palette category and a set of component types. If you are adding a component to an existing registration, you can either add the new component to the set in the existing statement, or add a new statement that calls *RegisterComponents*.

See Also

Creating a Unit File (see page 1317)

Deriving the Component Object (see page 1318)

Declaring a New Constructor (see page 1318)

3.2.1.8.16 Removing Dependencies

One quality that makes components usable is the absence of restrictions on what they can do at any point in their code. By their nature, components are incorporated into applications in varying combinations, orders, and contexts. You should design components that function in any situation, without preconditions.

An example of removing dependencies is the *Handle* property of *TWinControl*. If you have written Windows applications before, you know that one of the most difficult and error-prone aspects of getting a program running is making sure that you do not try to access a windowed control until you have created it by calling the *CreateWindow* API function. Delphi windowed controls relieve users from this concern by ensuring that a valid window handle is always available when needed. By using a property to represent the window handle, the control can check whether the window has been created; if the handle is not valid, the control creates a window and returns the handle. Thus, whenever an application's code accesses the *Handle* property, it is assured of getting a valid handle.

By removing background tasks like creating the window, Delphi components allow developers to focus on what they really want to do. Before passing a window handle to an API function, you do not need to verify that the handle exists or to create the window. The application developer can assume that things will work, instead of constantly checking for things that might go wrong.

Although it can take time to create components that are free of dependencies, it is generally time well spent. It not only spares application developers from repetition and drudgery, but it reduces your documentation and support burdens.

3.2.1.8.17 Registering Components (Introduction)

Before you can install your components in the IDE, you have to register them. Registration tells Delphi where to place the component on the Tool palette. You can also customize the way Delphi stores your components in the form file. For information on registering a component, see Registering components. (see page 1352)

3.2.1.8.18 Subclassing Windows Controls

In traditional Windows programming, you create custom controls by defining a new *window class* and registering it with Windows. The window class (which is similar to the *objects* or *classes* in object-oriented programming) contains information shared among instances of the same sort of control; you can base a new window class on an existing class, which is called *subclassing*. You then put your control in a dynamic-link library (DLL), much like the standard Windows controls, and provide an interface to it.

You can create a component "wrapper" around any existing window class. So if you already have a library of custom controls that you want to use in Delphi applications, you can create Delphi components that behave like your controls, and derive new controls from them just as you would with any other component.

For examples of the techniques used in subclassing Windows controls, see the components in the StdCtls unit that represent standard Windows controls, such as *TEdit*.

3.2.1.8.19 Creating a Component with the Component Wizard

The Component wizard simplifies the initial stages of creating a component. When you use the Component wizard, you need to specify:

- The class from which the component is derived.
- The class name for the new component.
- The Tool palette category where you want it to appear.

- The name of the unit in which the component is created.
- The search path where the unit is found.
- The name of the package in which you want to place the component.

The Component wizard performs the same tasks you would when creating a component manually:

- Creating a unit.
- Deriving the component.
- Registering the component.

The Component wizard cannot add components to an existing unit. You must add components to existing units manually.

To add a new component with the Component Wizard

1. To start the Component wizard, choose one of these two methods:

- Choose **Component** ▸ **New VCL Component**.
- Choose **File** ▸ **New** ▸ **Other**, goto the **Delphi Projects** ▸ **Delphi Files** page and double-click Component.

2. Fill in the fields in the Component wizard:

- In the Ancestor Type field, specify the class from which you are deriving your new component.
- In the Class Name field, specify the name of your new component class.
- In the Palette Page field, specify the category on the Tool palette on which you want the new component to be installed.
- In the Unit file name field, specify the name of the unit you want the component class declared in. If the unit is not on the search path, edit the search path in the Search Path field as necessary.

3. After you fill in the fields in the Component wizard, Click Install. To place the component in a new or existing package, click **Component** ▸ **Install** and use the dialog box that appears to specify a package.
4. Click OK. The IDE creates a new unit.

   Warning: If you derive a component from a class whose name begins with "custom" (such as *TCustomControl*), do not try to place the new component on a form until you have overridden any abstract methods in the original component. Delphi cannot create instance objects of a class that has abstract properties or methods.

   To see the source code for your unit, click **View** ▸ **Units...** (If the Component wizard is already closed, open the unit file in the Code editor by selecting **File** ▸ **Open**.) Delphi creates a new unit containing the class declaration and the *Register* procedure, and adds a **uses** clause that includes all the standard Delphi units.

   The unit looks like this:

```
unit MyControl;
    interface
    uses
      Windows, Messages, SysUtils, Types, Classes, Controls;
    type
      TMyControl = class(TCustomControl)
      private
      { Private declarations }
      protected
      { Protected declarations }
      public
      { Public declarations }
      published
      { Published declarations }
    end;
    procedure Register;
    implementation
    procedure Register;
    begin
      RegisterComponents('Samples', [TMyControl]); //In CLX, use a different page than
'Samples'
    end;
```

```
    end.
//header file
#ifndef NewComponentH
#define NewComponentH
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Controls.hpp>
#include <Classes.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TNewComponent : public TComponent
{
private:
protected:
public:
__fastcall TNewComponent(TComponent* Owner);
__published:
};
//---------------------------------------------------------------------------
#endif
//implementation file
#include <vcl.h>
#pragma hdrstop
#include "NewComponent.h"
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//
static inline void ValidCtrCheck(TNewComponent *)
{
new TNewComponent(NULL);
}
//---------------------------------------------------------------------------
__fastcall TNewComponent::TNewComponent(TComponent* Owner)
: TComponent(Owner)
{
}
//---------------------------------------------------------------------------
namespace Newcomponent
{
void __fastcall PACKAGE Register()
{
TComponentClass classes[1] = {__classid(TNewComponent)};
RegisterComponents("Samples", classes, 0); //In CLX use a different page than Samples
}
}
```

3.2.1.8.20 What Goes into a Component?

To make your components reliable parts of the Delphi environment, you need to follow certain conventions in their design. This section discusses the following topics:

- Removing dependencies (see page 1321)
- Setting properties (see page 1319)
- Encapsulating graphics (see page 1319)
- Registering components (see page 1321)

3.2.1.8.21 Making Source Files Available

Component writers should make all source files used by a component should be located in the same directory. These files

include source code files (.pas) and additional project files (.dfm/.xfm, .res, .rc, and .dcr).

The process of adding a component results in the creation of a number of files. These files are automatically put in directories specified in the IDE environment options (use the menu command **Tools ▸ Options**, navigate to the **Environment Options ▸ Delphi Options ▸ Library** page). The .lib files are placed in the DCP output directory. If adding the component entails creating a new package (as opposed to installing it into an existing package), the .bpl file is put in the BPL output directory.

3.2.1.8.22 Testing Installed Components

You can test the design-time behavior of a component after you install it on the Tool palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Tool palette. For information on testing components that have not yet been installed, see Testing uninstalled components (see page 1324).

Testing your components after installing allows you to debug the component that only generates design-time exceptions when dropped on a form.

Test an installed component using a second running instance of the IDE:

1. Choose **Project ▸ Options ▸ Debugger** and set the Debug Source Path to point to the component's source file.
2. Then select **Tools ▸ Options**. On the **Debugger Options ▸ Borland Debuggers ▸ Language Exceptions** page, enable the exceptions you want to track.
3. Open the component source file and set breakpoints.
4. Select **Run ▸ Parameters** and set the Host Application field to the name and location of the Delphi executable file.
5. In the Run Parameters dialog, click the Load button to start a second instance of Delphi.
6. Then drop the components to be tested on the form, which should break on your breakpoints in the source.

3.2.1.8.23 Testing Uninstalled Components

You can test the runtime behavior of a component before you install it on the Tool palette. This is particularly useful for debugging newly created components, but the same technique works with any component, whether or not it is on the Tool palette. For information on testing already installed components, see Testing installed components (see page 1324).

You test an uninstalled component by emulating the actions performed by Delphi when the component is selected from the palette and placed on a form.

To test an uninstalled component,

1. Add the name of component's unit to the form unit's **uses** clause.
2. Add an object field to the form to represent the component. This is one of the main differences between the way you add components and the way Delphi does it. You add the object field to the public part at the bottom of the form's type declaration. Delphi would add it above, in the part of the type declaration that it manages. Never add fields to the Delphi-managed part of the form's type declaration. The items in that part of the type declaration correspond to the items stored in the form file. Adding the names of components that do not exist on the form can render your form file invalid.
3. Attach a handler to the form's *OnCreate* event.
4. Construct the component in the form's *OnCreate* handler. When you call the component's constructor, you must pass a parameter specifying the owner of the component (the component responsible for destroying the component when the time comes). You will nearly always pass *Self* as the owner. In a method, *Self* is a reference to the object that contains the method. In this case, in the form's *OnCreate* handler, *Self* refers to the form.
5. Assign the *Parent* property. Setting the *Parent* property is always the first thing to do after constructing a control. The parent is the component that contains the control visually; usually it is the form on which the control appears, but it might be a group box or panel. Normally, you'll set *Parent* to *Self*, that is, the form. Always set *Parent* before setting other properties of the

control.

Warning: If your component is not a control (that is, if *TControl* is not one of its ancestors), skip this step. If you accidentally set the form's *Parent* property (instead of the component's) to *Self*, you can cause an operating-system problem.

6. Set any other component properties as desired.

3.2.1.9 Making a control data aware

Topics

Name	Description
Adding the Data Link (see page 1327)	The connection between a control and a database is handled by a class called a *data link*. The data link class that connects a control with a single field in a database is *TFieldDataLink*. There are also data links for entire tables. A data-aware control *owns* its data link class. That is, the control has the responsibility for constructing and destroying the data link. For details on management of owned classes, see Creating a graphic control (see page 1220)
Adding the ReadOnly property (see page 1328)	By adding a *ReadOnly* property, you will provide a way to make the control read-only at design time. When that property is set to *True*, you can make all cells in the control unable to be selected.
Allowing Needed Updates (see page 1329)	The read-only calendar uses the *SelectCell* method for all kinds of changes, including setting the *Row* and *Col* properties. The *UpdateCalendar* method sets *Row* and *Col* every time the date changes, but because *SelectCell* disallows changes, the selection remains in place, even though the date changes. To get around this absolute prohibition on changes, you can add an internal Boolean flag to the calendar, and permit changes when that flag is set to *True*:
Changing the Default Value of FReadOnly (see page 1330)	Because this is a data editing control, the *ReadOnly* property should be set to *False* by default. To make the *ReadOnly* property *False*, change the value of *FReadOnly* in the constructor:
Creating a Data Browsing Control (see page 1330)	Creating a data-aware calendar control, whether it is a read-only control or one in which the user can change the underlying data in the dataset, involves the following steps: • Creating and registering the component (see page 1331). • Adding the data link (see page 1327). • Responding to data changes (see page 1334).
Creating a Data Editing Control (see page 1330)	When you create a data editing control, you create and register the component and add the data link just as you do for a data browsing control. You also respond to data changes in the underlying field in a similar manner, but you must handle a few more issues. For example, you probably want your control to respond to both key and mouse events. Your control must respond when the user changes the contents of the control. When the user exits the control, you want the changes made in the control to be reflected in the dataset. The data editing... more (see page 1330)

Creating and registering the data-aware component (see page 1331)	You create every component the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. This process is outlined in Creating a new component (see page 1317). For this example, follow the general procedure for creating a component, with these specifics: • Call the component's unit *DBCal*. • Derive a new component class called *TDBCalendar*, descended from the component *TSampleCalendar*. The section Customizing a grid (see page 1264) shows you how to create the *TSampleCalendar* component. • Register *TDBCalendar* on the Samples page of the Tool palette. The resulting unit descending from *TCustomGrid* in a... more (see page 1331)
Declaring the Access Properties for a Data-aware Control (see page 1331)	Every data-aware control has a *DataSource* property that specifies which data source class in the application provides the data to the control. In addition, a control that accesses a single field needs a *DataField* property to specify that field in the data source. Unlike the access properties for the owned classes in the example in Creating a graphic control (see page 1220) these access properties do not provide access to the owned classes themselves, but rather to corresponding properties in the owned class. That is, you will create properties that enable the control and its data link to share the same data source... more (see page 1331)
Declaring the Class Field (see page 1332)	A component needs a field for each of its owned classes, as explained in Declaring the class fields (see page 1222). In this case, the calendar needs a field of type *TFieldDataLink* for its data link. Declare a field for the data link in the calendar:
Handling Mouse-down and Key-down Messages (see page 1332)	When the user of the control begins interacting with it, the control receives either mouse-down messages (*WM_LBUTTONDOWN*, *WM_MBUTTONDOWN*, or *WM_RBUTTONDOWN*) or a key-down message (*WM_KEYDOWN*) from Windows. To enable a control to respond to these messages, you must write handlers that respond to these messages. • Responding to mouse-down messages (see page 1336). • Responding to key-down messages (see page 1335).
Initializing the Data Link (see page 1332)	A data-aware control needs access to its data link throughout its existence, so it must construct the data link object as part of its own constructor, and destroy the data link object before it is itself destroyed. Override the *Create* and *Destroy* methods of the calendar to construct and destroy the datalink object, respectively:
Making a Control Data Aware (see page 1333)	When working with database connections, it is often convenient to have controls that are *data aware*. That is, the application can establish a link between the control and some part of a database. Delphi includes data-aware labels, edit boxes, list boxes, combo boxes, lookup controls, and grids. You can also make your own controls data aware. For more information about using data-aware controls, see Using data controls (see page 1778) . There are several degrees of data awareness. The simplest is read-only data awareness, or *data browsing*, the ability to reflect the current state of a database. More complicated is editable... more (see page 1333)
Making the Control Read-only (see page 1333)	Because this data calendar will be read-only with respect to the data, it makes sense to make the control itself read-only, so users will not make changes within the control and expect them to be reflected in the database. Making the calendar read-only involves: • Adding the ReadOnly property. (see page 1328) • Allowing needed updates. (see page 1329) **Note:** Note that if you started with the *TCalendar* component from Delphi's Samples page instead of *TSampleCalendar*, it already has a *ReadOnly* property, so you can skip these steps.

Modifying the Change Method (see page 1334)	The *Change* method of the *TDBCalendar* is called whenever a new date value is set. *Change* calls the *OnChange* event handler, if one exists. The component user can write code in the *OnChange* event handler to respond to changes in the date. When the calendar date changes, the underlying dataset should be notified that a change has occurred. You can do that by overriding the *Change* method and adding one more line of code.
Responding to Data Changes (see page 1334)	Once a control has a data link and properties to specify the data source and data field, it needs to respond to changes in the data in that field, either because of a move to a different record or because of a change made to that field. Data link classes all have events named *OnDataChange*. When the data source indicates a change in its data, the data link object calls any event handler attached to its *OnDataChange* event. To update a control in response to data changes, attach a handler to the data link's *OnDataChange* event. In this case,... more (see page 1334)
Responding to Key-down Messages (see page 1335)	A *KeyDown* method is a protected method for a control's *OnKeyDown* event. The control itself calls *KeyDown* in response to a Windows key-down message. When overriding the inherited *KeyDown* method, you can include code that provides other responses in addition to calling the *OnKeyDown* event.
Responding to Mouse-down Messages (see page 1336)	A *MouseDown* method is a protected method for a control's *OnMouseDown* event. The control itself calls *MouseDown* in response to a Windows mouse-down message. When you override the inherited *MouseDown* method, you can include code that provides other responses in addition to calling the *OnMouseDown* event. To override *MouseDown*, add the *MouseDown* method to the *TDBCalendar* class:
Updating the Dataset (see page 1337)	So far, a change within the data-aware control has changed values in the field data link class. The final step in creating a data editing control is to update the dataset with the new value. This should happen after the person changing the value in the data-aware control exits the control by clicking outside the control or pressing the `Tab` key. **Note:** VCL applications define message control IDs for operations on controls. For example, the *CM_EXIT* message is sent to the control when the user exits the control. You can write message handlers that respond to the message. In this... more (see page 1337)
Updating the Field Data Link Class (see page 1338)	There are two types of data changes: • A change in a field value that must be reflected in the data-aware control. • A change in the data-aware control that must be reflected in the field value. The *TDBCalendar* component already has a *DataChange* method that handles a change in the field's value in the dataset by assigning that value to the *CalendarDate* property. The *DataChange* method is the handler for the *OnDataChange* event. So the calendar component can handle the first type of data change. Similarly, the field data link class also has an *OnUpdateData* event that occurs as the user... more (see page 1338)

3.2.1.9.1 Adding the Data Link

The connection between a control and a database is handled by a class called a *data link*. The data link class that connects a control with a single field in a database is *TFieldDataLink*. There are also data links for entire tables.

A data-aware control *owns* its data link class. That is, the control has the responsibility for constructing and destroying the data link. For details on management of owned classes, see Creating a graphic control (see page 1220)

Establishing a data link as an owned class requires these three steps:

1. Declaring the class field (see page 1332).
2. Declaring the access properties (see page 1331).

3. Initializing the data link (see page 1332).

3.2.1.9.2 Adding the ReadOnly property

By adding a *ReadOnly* property, you will provide a way to make the control read-only at design time. When that property is set to *True*, you can make all cells in the control unable to be selected.

To add the ReadOnly property, follow these steps:

1. Add the property declaration and a **private** field to hold the value:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FReadOnly: Boolean;                                    { field for internal storage }
  public
    constructor Create(AOwner: TComponent); override;     { must override to set default }
  published
    property ReadOnly: Boolean read FReadOnly write FReadOnly default True;
  end;
.
.
.
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                    { always call the inherited constructor! }
  FReadOnly := True;                                          { set the default value }
end;
//header file
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    bool FReadOnly;                                  // field for internal storage
protected:
public:
    virtual __fastcall TDBCalendar(TComponent* Owner);
__published:
    __property ReadOnly = {read=FReadOnly, write=FReadOnly, default=true};
};
//implementation file:
virtual __fastcall TDBCalendar::TDBCalendar(TComponent* Owner) :
  TSampleCalendar(Owner)
{
  FReadOnly = true;                                   // sets the default value
}
```

2. Override the *SelectCell* method to disallow selection if the control is read-only. Use of *SelectCell* is explained in Excluding blank cells (see page 1265).

```
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if FReadOnly then Result := False                          { cannot select if read only }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;
bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
  if (FReadOnly) return false;                           // can't select if read only
  return TSampleCalendar::SelectCell(ACol, ARow);     // otherwise, use inherited method
}
```

Remember to add the declaration of *SelectCell* to the type declaration of *TDBCalendar*, and append the **override** directive.

If you now add the calendar to a form, you will find that the component ignores clicks and keystrokes. It also fails to update the selection position when you change the date.

3.2.1.9.3 Allowing Needed Updates

The read-only calendar uses the *SelectCell* method for all kinds of changes, including setting the *Row* and *Col* properties. The *UpdateCalendar* method sets *Row* and *Col* every time the date changes, but because *SelectCell* disallows changes, the selection remains in place, even though the date changes.

To get around this absolute prohibition on changes, you can add an internal Boolean flag to the calendar, and permit changes when that flag is set to *True*:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FUpdating: Boolean;                                    { private flag for internal use }
  protected
    function SelectCell(ACol, ARow: Longint): Boolean; override;
  public
    procedure UpdateCalendar; override;                { remember the override directive }
  end;
.
.
.
function TDBCalendar.SelectCell(ACol, ARow: Longint): Boolean;
begin
  if (not FUpdating) and FReadOnly then Result := False       { allow select if updating }
  else Result := inherited SelectCell(ACol, ARow);     { otherwise, use inherited method }
end;
procedure TDBCalendar.UpdateCalendar;
begin
  FUpdating := True;                                          { set flag to allow updates }
  try
    inherited UpdateCalendar;                                          { update as usual }
  finally
    FUpdating := False;                                       { always clear the flag }
  end;
end;
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    .
    .
    .
    bool FUpdating;                                    // private flag for internal use
protected:
    virtual bool __fastcall SelectCell(long ACol, long ARow);
public:
    .
    .
    .
    virtual void __fastcall UpdateCalendar();
    .
    .
    .
};
bool __fastcall TDBCalendar::SelectCell(long ACol, long ARow)
{
    if (!FUpdating && FReadOnly) return false;          // can't select if read only
    return TSampleCalendar::SelectCell(ACol, ARow);     // otherwise, use inherited method
}
void __fastcall TDBCalendar::UpdateCalendar()
{
  FUpdating=true;                                       // set flag to allow updates
  try
  {
     TSampleCalendar::UpdateCalendar();              // update as usual
```

```
  }
  catch(...)
  {
    FUpdating = false;
    throw;
  }
  FUpdating = false;                                    // always clear the flag
}
```

The calendar still disallows user changes, but now correctly reflects changes made in the date by changing the date properties. Now that you have a true read-only calendar control, you are ready to add the data browsing ability.

3.2.1.9.4 Changing the Default Value of FReadOnly

Because this is a data editing control, the *ReadOnly* property should be set to *False* by default. To make the *ReadOnly* property *False*, change the value of *FReadOnly* in the constructor:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  .
  .
  .
  FReadOnly := False;  { set the default value }
  .
  .
  .
end;
__fastcall TDBCalendar::TDBCalendar (TComponent* Owner) : TSampleCalendar(Owner)
{
  FReadOnly = false;             // set the default value
  .
  .
  .
}
```

3.2.1.9.5 Creating a Data Browsing Control

Creating a data-aware calendar control, whether it is a read-only control or one in which the user can change the underlying data in the dataset, involves the following steps:

- Creating and registering the component (see page 1331).
- Adding the data link (see page 1327).
- Responding to data changes (see page 1334).

3.2.1.9.6 Creating a Data Editing Control

When you create a data editing control, you create and register the component and add the data link just as you do for a data browsing control. You also respond to data changes in the underlying field in a similar manner, but you must handle a few more issues.

For example, you probably want your control to respond to both key and mouse events. Your control must respond when the user changes the contents of the control. When the user exits the control, you want the changes made in the control to be reflected in the dataset.

The data editing control described here is the same calendar control described in Creating a data browsing control (see page 1330). The control is modified so that it can edit as well as view the data in its linked field.

Modifying the existing control to make it a data editing control involves:

- Changing the default value of FReadOnly. (see page 1330)
- Handling mouse-down and key-down messages. (see page 1332)
- Updating the field data link class. (see page 1338)
- Modifying the Change method. (see page 1334)
- Updating the dataset. (see page 1337)

3.2.1.9.7 Creating and registering the data-aware component

You create every component the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. This process is outlined in Creating a new component (see page 1317).

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *DBCal*.
- Derive a new component class called *TDBCalendar*, descended from the component *TSampleCalendar*. The section Customizing a grid (see page 1264) shows you how to create the *TSampleCalendar* component.
- Register *TDBCalendar* on the Samples page of the Tool palette.

The resulting unit descending from *TCustomGrid* in a VCL application should look like this:

```
unit CalSamp;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Grids;
type
  TSampleCalendar = class(TCustomGrid)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TSampleCalendar]);
end;
end.
```

If you install the calendar component now, you will find that it appears on the Samples page. The only properties available are the most basic control properties. The next step is to make some of the more specialized properties available to users of the calendar.

Note: While you can install the sample calendar component you have just compiled, do not try to place it on a form yet. The *TCustomGrid* component has an abstract *DrawCell* method that must be redeclared before instance objects can be created. Overriding the DrawCell method is described in Filling in the cells (see page 1265).

3.2.1.9.8 Declaring the Access Properties for a Data-aware Control

Every data-aware control has a *DataSource* property that specifies which data source class in the application provides the data to the control. In addition, a control that accesses a single field needs a *DataField* property to specify that field in the data source.

Unlike the access properties for the owned classes in the example in Creating a graphic control (see page 1220) these access properties do not provide access to the owned classes themselves, but rather to corresponding properties in the owned class. That is, you will create properties that enable the control and its data link to share the same data source and field.

Declare the *DataSource* and *DataField* properties and their implementation methods, then write the methods as "pass-through" methods to the corresponding properties of the data link class.

See Also

Initializing the Data Link (see page 1332)

3.2.1.9.9 Declaring the Class Field

A component needs a field for each of its owned classes, as explained in Declaring the class fields (see page 1222). In this case, the calendar needs a field of type *TFieldDataLink* for its data link.

Declare a field for the data link in the calendar:

```
type
  TDBCalendar = class(TSampleCalendar)
  private
    FDataLink: TFieldDataLink;
  .
  .
  .
  end;
```

```
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    TFieldDataLink *FDataLink;
    .
    .
    .
};
#include <DB.hpp>
#include <DBTables.hpp>
```

Before you can compile the application, you need to add DB and DBCtrls to the unit's **uses** clause.

3.2.1.9.10 Handling Mouse-down and Key-down Messages

When the user of the control begins interacting with it, the control receives either mouse-down messages (*WM_LBUTTONDOWN*, *WM_MBUTTONDOWN*, or *WM_RBUTTONDOWN*) or a key-down message (*WM_KEYDOWN*) from Windows. To enable a control to respond to these messages, you must write handlers that respond to these messages.

- Responding to mouse-down messages (see page 1336).
- Responding to key-down messages (see page 1335).

3.2.1.9.11 Initializing the Data Link

A data-aware control needs access to its data link throughout its existence, so it must construct the data link object as part of its own constructor, and destroy the data link object before it is itself destroyed.

Override the *Create* and *Destroy* methods of the calendar to construct and destroy the datalink object, respectively:

```
type
  TDBCalendar = class(TSampleCalendar)
  public                                  { constructors and destructors are always public }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    .
    .
    .
  end;
.
.
.
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);                  { always call the inherited constructor first }
  FDataLink := TFieldDataLink.Create;                        { construct the datalink object }
```

```
  FDataLink.Control := self;                {let the datalink know about the calendar }
  FReadOnly := True;                                                { this is already here }
end;
destructor TDBCalendar.Destroy;
begin
  FDataLink.Free;                                    { always destroy owned objects first... }
  inherited Destroy;                                     { ...then call inherited destructor }
end;
class PACKAGE TDBCalendar : public TSampleCalendar
{
public:
    virtual __fastcall TDBCalendar(TComponent *Owner);
    __fastcall ~TDBCalendar();
};
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner) : TSampleCalendar(Owner)
{
    FReadOnly = true;
    FDataLink = new TFieldDataLink();
    FDataLink->Control = this;
}
__fastcall TDBCalendar::~TDBCalendar()
{
    FDataLink->Control = NULL;
    FDataLink->OnUpdateData = NULL;
    delete FDataLink;
}
```

Now you have a complete data link, but you have not yet told the control what data it should read from the linked field. The next section (see page 1334) explains how to do that.

See Also

Declaring the Access Properties for a Data-aware Control (see page 1331)

3.2.1.9.12 Making a Control Data Aware

When working with database connections, it is often convenient to have controls that are *data aware*. That is, the application can establish a link between the control and some part of a database. Delphi includes data-aware labels, edit boxes, list boxes, combo boxes, lookup controls, and grids. You can also make your own controls data aware. For more information about using data-aware controls, see Using data controls (see page 1778) .

There are several degrees of data awareness. The simplest is read-only data awareness, or *data browsing*, the ability to reflect the current state of a database. More complicated is editable data awareness, or *data editing*, where the user can edit the values in the database by manipulating the control. Note also that the degree of involvement with the database can vary, from the simplest case, a link with a single field, to more complex cases, such as multiple-record controls.

This section first illustrates the simplest case, making a read-only control that links to a single field in a dataset. The specific control used will be the *TSampleCalendar* calendar created in Customizing a grid (see page 1264) You can also use the standard calendar control on the Samples page of the Tool palette, *TCalendar*.

The section then continues with an explanation of how to make the new data browsing control a data editing control.

3.2.1.9.13 Making the Control Read-only

Because this data calendar will be read-only with respect to the data, it makes sense to make the control itself read-only, so users will not make changes within the control and expect them to be reflected in the database.

Making the calendar read-only involves:

- Adding the ReadOnly property. (see page 1328)

- Allowing needed updates. (see page 1329)

  Note: Note that if you started with the *TCalendar* component from Delphi's Samples page instead of *TSampleCalendar*, it already has a *ReadOnly* property, so you can skip these steps.

3.2.1.9.14 Modifying the Change Method

The *Change* method of the *TDBCalendar* is called whenever a new date value is set. *Change* calls the *OnChange* event handler, if one exists. The component user can write code in the *OnChange* event handler to respond to changes in the date.

When the calendar date changes, the underlying dataset should be notified that a change has occurred. You can do that by overriding the *Change* method and adding one more line of code.

These are the steps to follow:

1. Add a new *Change* method to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure Change; override;
   .
   .
   .
  end;
class PACKAGE TDBCalendar : public TSampleCalendar
{
protected:
    virtual void __fastcall Change();
     .
     .
     .
};
```

2. Write the *Change* method, calling the *Modified* method that informs the dataset the data has changed, then call the inherited *Change* method:

```
procedure TDBCalendar.Change;
begin
  FDataLink.Modified;                  { call the Modified method }
  inherited Change;                    { call the inherited Change method }
end;
void __fastcall TDBCalendar::Change()
{
  if (FDataLink != NULL)
    FDataLink->Modified();               // call the Modified method
  TSampleCalendar::Change();             // call the inherited Change method
}
```

3.2.1.9.15 Responding to Data Changes

Once a control has a data link and properties to specify the data source and data field, it needs to respond to changes in the data in that field, either because of a move to a different record or because of a change made to that field.

Data link classes all have events named *OnDataChange*. When the data source indicates a change in its data, the data link object calls any event handler attached to its *OnDataChange* event.

To update a control in response to data changes, attach a handler to the data link's *OnDataChange* event.

In this case, you will add a method to the calendar, then designate it as the handler for the data link's *OnDataChange*.

Declare and implement the *DataChange* method, then assign it to the data link's *OnDataChange* event in the constructor. In the destructor, detach the *OnDataChange* handler before destroying the object.

3.2.1.9.16 Responding to Key-down Messages

A *KeyDown* method is a protected method for a control's *OnKeyDown* event. The control itself calls *KeyDown* in response to a Windows key-down message. When overriding the inherited *KeyDown* method, you can include code that provides other responses in addition to calling the *OnKeyDown* event.

To override KeyDown, follow these steps:

1. Add a *KeyDown* method to the *TDBCalendar* class:

```
type
  TDBCalendar = class(TSampleCalendar);
    .
    .
    .
  protected
    procedure KeyDown(var Key: Word; Shift: TShiftState; X: Integer; Y: Integer);
      override;
    .
    .
    .
  end;
class PACKAGE TDBCalendar : public TSampleCalendar
{
    .
    .
    .
protected:
    virtual void __fastcall KeyDown(unsigned short &Key,  TShiftState Shift);
    .
    .
    .
};
```

2. Implement the *KeyDown* method:

```
procedure KeyDown(var Key: Word; Shift: TShiftState);
var
  MyKeyDown: TKeyEvent;
begin
  if not ReadOnly and (Key in [VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT, VK_END,
    VK_HOME, VK_PRIOR, VK_NEXT]) and FDataLink.Edit then
    inherited KeyDown(Key, Shift)
  else
  begin
    MyKeyDown := OnKeyDown;
    if Assigned(MyKeyDown) then MyKeyDown(Self, Key, Shift);
  end;
end;
void __fastcall TDBCalendar::KeyDown(unsigned short &Key,  TShiftState Shift)
{
  TKeyEvent MyKeyDown;                                  // declare event type
  Set<unsigned short,0,8> keySet;
  keySet = keySet << VK_UP << VK_DOWN << VK_LEFT      // assign virtual keys to set
    << VK_RIGHT << VK_END << VK_HOME << VK_PRIOR << VK_NEXT;
  if (!FReadOnly &&                                  // if control is not read only...
      (keySet.Contains(Key)) &&                       // ...and key is in the set...
      FDataLink->Edit() )                             // ...and field is in edit mode
  {
    TCustomGrid::KeyDown(Key, Shift);           // call the inherited KeyDown method
  }
  else
  {
    MyKeyDown = OnKeyDown;                              // assign OnKeyDown event
    if (MyKeyDown != NULL) MyKeyDown(this,Key,Shift); // execute code in...
```

```
  }                                                    // ...OnKeyDown event handler
}
```

When *KeyDown* responds to a mouse-down message, the inherited *KeyDown* method is called only if the control's *ReadOnly* property is *False*, the key pressed is one of the cursor control keys, and the data link object is in edit mode, which means the field can be edited. If the field cannot be edited or some other key is pressed, the code the programmer put in the *OnKeyDown* event handler, if one exists, is executed.

3.2.1.9.17 Responding to Mouse-down Messages

A *MouseDown* method is a protected method for a control's *OnMouseDown* event. The control itself calls *MouseDown* in response to a Windows mouse-down message. When you override the inherited *MouseDown* method, you can include code that provides other responses in addition to calling the *OnMouseDown* event.

To override *MouseDown*, add the *MouseDown* method to the *TDBCalendar* class:

```
type
  TDBCalendar = class(TSampleCalendar);
    .
    .
    .
  protected
    procedure MouseDown(Button: TButton, Shift: TShiftState, X: Integer, Y: Integer);
      override;
    .
    .
    .
  end;
procedure TDBCalendar.MouseDown(Button: TButton; Shift: TShiftState; X, Y: Integer);
var
  MyMouseDown: TMouseEvent;
begin
  if not ReadOnly and FDataLink.Edit then
    inherited MouseDown(Button, Shift, X, Y)
  else
  begin
    MyMouseDown := OnMouseDown;
    if Assigned(MyMouseDown then MyMouseDown(Self, Button, Shift, X, Y);
  end;
end;
//header file
class PACKAGE TDBCalendar : public TSampleCalendar
{
.
.
.
protected:
    virtual void __fastcall MouseDown(TMouseButton Button,  TShiftState Shift, int X,
int Y);
    .
    .
    .
};
//implmentation file
void __fastcall TDBCalendar::MouseDown(TMouseButton Button,  TShiftState Shift, int  X,   int
Y)
{
  TMouseEvent MyMouseDown;                              // declare event type
  if (!FReadOnly && FDataLink->Edit())                  // if the field can be edited
    TSampleCalendar::MouseDown(Button, Shift, X, Y);   // call the inherited MouseDown
  else
  {
    MyMouseDown = OnMouseDown;                          // assign OnMouseDown event
    if (MyMouseDown != NULL) MyMouseDown(this, Button,  // execute code in the...
```

```
      Shift, X, Y);                                  // ...OnMouseDown event handler
  }
}
```

When *MouseDown* responds to a mouse-down message, the inherited *MouseDown* method is called only if the control's *ReadOnly* property is *False* and the data link object is in edit mode, which means the field can be edited. If the field cannot be edited, the code the programmer put in the *OnMouseDown* event handler, if one exists, is executed.

3.2.1.9.18 Updating the Dataset

So far, a change within the data-aware control has changed values in the field data link class. The final step in creating a data editing control is to update the dataset with the new value. This should happen after the person changing the value in the data-aware control exits the control by clicking outside the control or pressing the Tab key.

Note: VCL applications define message control IDs for operations on controls. For example, the *CM_EXIT* message is sent to the control when the user exits the control. You can write message handlers that respond to the message. In this case, when the user exits the control, the *CMExit* method, the message handler for *CM_EXIT*, responds by updating the record in the dataset with the changed values in the field data link class. For more information about message handlers, see Handling messages and system notifications. (see page 1308)

To update the dataset within a message handler, follow these steps:

1. Add the message handler to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure CMExit(var Message: TWMNoParams); message CM_EXIT;
   .
   .
   .
  end;
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall CMExit(TWMNoParams Message);
BEGIN_MESSAGE_MAP
  MESSAGE_HANDLER(CM_EXIT, TWMNoParams, CMExit)
END_MESSAGE_MAP
};
```

2. Implement the *CMExit* method so it looks like this:

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;                          { tell data link to update database }
  except
    on Exception do SetFocus;                       { if it failed, don't let focus leave }
 end;
  inherited;
end;
void __fastcall TDBCalendar::CMExit(TWMNoParams &Message)
{
  try
  {
    FDataLink.UpdateRecord();          // tell data link to update database
  }
  catch(...)
  {
    SetFocus();                        // if it failed, don't let focus leave
    throw;
  }
```

```
}
```

To update the dataset when the user exits the control, follow these steps:

1. Add an override for the *DoExit* method to the *TDBCalendar* component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure DoExit; override;
   .
   .
   .
  end;
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    DYNAMIC void __fastcall DoExit(void);
   .
   .
   .
};
```

2. Implement the *DoExit* method so it looks like this:

```
procedure TDBCalendar.CMExit(var Message: TWMNoParams);
begin
  try
    FDataLink.UpdateRecord;                          { tell data link to update database }
  except
    on Exception do SetFocus;                       { if it failed, don't let focus leave }
 end;
  inherited;                           { let the inherited method generate an OnExit event }
end;
void __fastcall TDBCalendar::DoExit(void)
{
  try
  {
    FDataLink.UpdateRecord();          // tell data link to update database
  }
  catch(...)
  {
    SetFocus();                        // if it failed, don't let focus leave
    throw;
  }
  TCustomGrid::DoExit(); // let the inherited method generate an OnExit event
}
```

3.2.1.9.19 Updating the Field Data Link Class

There are two types of data changes:

- A change in a field value that must be reflected in the data-aware control.
- A change in the data-aware control that must be reflected in the field value.

The *TDBCalendar* component already has a *DataChange* method that handles a change in the field's value in the dataset by assigning that value to the *CalendarDate* property. The *DataChange* method is the handler for the *OnDataChange* event. So the calendar component can handle the first type of data change.

Similarly, the field data link class also has an *OnUpdateData* event that occurs as the user of the control modifies the contents of the data-aware control. The calendar control has a *UpdateData* method that becomes the event handler for the *OnUpdateData* event. *UpdateData* assigns the changed value in the data-aware control to the field data link.

To reflect a change made to the value in the calendar in the field value:

1. Add an *UpdateData* method to the private section of the calendar component:

```
type
  TDBCalendar = class(TSampleCalendar);
  private
    procedure UpdateData(Sender: TObject);
   .
   .
   .
  end;
class PACKAGE TDBCalendar : public TSampleCalendar
{
private:
    void __fastcall UpdateData(TObject *Sender);
};
```

2. Implement the *UpdateData* method:

```
procedure UpdateData(Sender: TObject);
begin
  FDataLink.Field.AsDateTime := CalendarDate;        { set field link to calendar date }
end;
void __fastcall TDBCalendar::UpdateData( TObject* Sender)
{
    FDataLink->Field->AsDateTime = CalendarDate;     // set field link to calendar date
}
```

3. Within the constructor for *TDBCalendar*, assign the *UpdateData* method to the *OnUpdateData* event:

```
constructor TDBCalendar.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FReadOnly := True;
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  FDataLink.OnUpdateData := UpdateData;
end;
__fastcall TDBCalendar::TDBCalendar(TComponent* Owner)
  : TSampleCalendar(Owner)
{
  FDataLink = new TFieldDataLink();       // this was already here
  FDataLink->OnDataChange = DataChange;   // this was here too
  FDataLink->OnUpdateData = UpdateData;   // assign UpdateData to the OnUpdateData event
}
```

3.2.1.10 Making components available at design time

Topics

Name	Description
Adding Clipboard Formats (see page 1344)	By default, when a user chooses Copy while a component is selected in the IDE, the component is copied in Delphi's internal format. It can then be pasted into another form or data module. Your component can copy additional formats to the Clipboard by overriding the *Copy* method. For example, the following *Copy* method allows a *TImage* component to copy its picture to the Clipboard. This picture is ignored by the Delphi IDE, but can be pasted into other applications.
Adding Component Editors (see page 1344)	Component editors determine what happens when the component is double-clicked in the designer and add commands to the context menu that appears when the component is right-clicked. They can also copy your component to the Windows clipboard in custom formats. If you do not give your components a component editor, Delphi uses the default component editor. The default component editor is implemented by the class *TDefaultEditor*. *TDefaultEditor* does not add any new items to a component's context menu. When the component is double-clicked, *TDefaultEditor* searches the properties of the component and generates (or navigates to) the first event handler... more (see page 1344)

Adding Items to the Context Menu (see page 1345)	When the user right-clicks the component, the *GetVerbCount* and *GetVerb* methods of the component editor are called to build context menu. You can override these methods to add commands (verbs) to the context menu. Adding items to the context menu requires the steps: • Specifying menu items (see page 1355) • Implementing commands (see page 1349)
Adding Property Editors (see page 1345)	The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type. At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing.... more (see page 1345)
Changing the Double-click Behavior (see page 1345)	When the component is double-clicked, the *Edit* method of the component editor is called. By default, the *Edit* method executes the first command added to the context menu. Thus, in the previous example (see page 1349), double-clicking the component executes the *DoThis* command. While executing the first command is usually a good idea, you may want to change this default behavior. For example, you can provide an alternate behavior if • you are not adding any commands to the context menu. • you want to display a dialog that combines several commands when the component is double-clicked. Override the *Edit* method to specify a... more (see page 1345)
Compiling Components into Packages (see page 1346)	Once your components are registered, you must compile them as packages before they can be installed in the IDE. A package can contain one or several components as well as custom property editors. For more information about packages, see Working with packages and components (see page 2211) . To create and compile a package, see Creating and editing packages (see page 2213) . Put the source-code units for your custom components in the package's Contains list. If your components depend on other packages, include those packages in the Requires list. To install your components in the IDE, see Installing component packages (see page 2217).
Creating the Entries (see page 1346)	To make your component's Help integrate seamlessly with the Help for the rest of the components in the library, observe the following conventions: **Each component should have a Help topic:** The component topic should show which unit the component is declared in and briefly describe the component. The component topic should link to secondary windows that describe the component's position in the object hierarchy and list all of its properties, events, and methods. Application developers access this topic by selecting the component on a form and pressing `F1`. For an example of a component topic, place any component on... more (see page 1346)
Creating the Help File (see page 1347)	You can use any tool you want to create the source file for a Windows Help file (in .rtf format). Delphi includes the Microsoft Help Workshop, which compiles your Help files and provides an online Help authoring guide. You can find complete information about creating Help files in the online guide for Help Workshop. Composing Help files for components consists of the steps: • Creating the entries (see page 1346). • Making component Help context-sensitive (see page 1350).
Declaring the Register Procedure (see page 1347)	Registration involves writing a single procedure in the unit, which must have the name *Register*. The *Register* procedure must appear in the interface part of the unit, and (unlike the rest of Delphi) its name is case-sensitive. **Note:** Although Delphi is a case insensitive language, the Register procedure is case sensitive and must be spelled with an uppercase R. The following code shows the outline of a simple unit that creates and registers new components:

Deriving a Property-editor Class (see page 1348)	Both the component library define several kinds of property editors, all of which descend from *TPropertyEditor*. When you create a property editor, your property-editor class can either descend directly from *TPropertyEditor* or indirectly through one of the property-editor classes described in the table below. The classes in the DesignEditors unit can be used for VCL applications. **Note:** All that is absolutely necessary for a property editor is that it descend from *TBasePropertyEditor* and that it support the *IProperty* interface. *TPropertyEditor*, however, provides a default implementation of the *IProperty* interface. The list in the table below is not... more (see page 1348)
Editing the Property as a Whole (see page 1349)	You can optionally provide a dialog box in which the user can visually edit a property. The most common use of property editors is for properties that are themselves classes. An example is the *Font* property, for which the user can open a font dialog box to choose all the attributes of the font at once. To provide a whole-property editor dialog box, override the property-editor class's *Edit* method. *Edit* methods use the same Get and Set methods used in writing *GetValue* and *SetValue* methods. In fact, an *Edit* method calls both a Get method and a Set method. Because... more (see page 1349)
Implementing Commands (see page 1349)	When the command provided by *GetVerb* is selected in the designer, the *ExecuteVerb* method is called. For every command you provide in the *GetVerb* method, implement an action in the *ExecuteVerb* method. You can access the component that is being edited using the *Component* property of the editor. For example, the following *ExecuteVerb* method implements the commands for the *GetVerb* method in the previous (see page 1355) example.
Making Component Help Context-sensitive (see page 1350)	Each component, property, method, and event topic must have an A footnote. The A footnote is used to display the topic when the user selects a component and presses `F1`, or when a property or event is selected in the Object Inspector and the user presses `F1`. The A footnotes must follow certain naming conventions: If the Help topic is for a component, the A footnote consists of two entries separated by a semicolon using this syntax:
Making Components Available at Design Time: Overview (see page 1350)	Making your components available at design time requires several steps: • Registering components (see page 1352) • Providing Help for your component (see page 1351) • Adding property editors (see page 1345) • Adding component editors (see page 1344) • Compiling components into packages (see page 1346) Not all these steps apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only steps that are always necessary are registration and compilation. Once your components have been registered and compiled into packages, they can be distributed to other developers and installed in the IDE. For information on installing packages in the IDE, see Installing component packages (see page 2217).... more (see page 1350)
Property Categories (see page 1350)	In the IDE, the Object Inspector lets you selectively hide and display properties based on property categories. The properties of new custom components can be fit into this scheme by registering properties in categories. Do this at the same time you register the component by calling *RegisterPropertyInCategory* or *RegisterPropertiesInCategory*. Use *RegisterPropertyInCategory* to register a single property. Use *RegisterPropertiesInCategory* to register multiple properties in a single function call. These functions are defined in the unit DesignIntf. Note that it is not mandatory that you register properties or that you register all of the properties of a custom component... more (see page 1350)
Specifying Property Categories (see page 1351)	When you register properties in a category, you can use any string you want as the name of the category. If you use a string that has not been used before, the Object Inspector generates a new property category class with that name. You can also, however, register properties into one of the categories that are built-in. The built-in property categories are described in the following table: ***Property categories***

Providing Help for Your Component (see page 1351)	When you select a standard component on a form, or a property or event in the Object Inspector, you can press F1 to get Help on that item. You can provide developers with the same kind of documentation for your components if you create the appropriate Help files. You can provide a small Help file to describe your components, and your Help file becomes part of the user's overall Delphi Help system. See the section Creating the Help file (see page 1347) for information on how to compose the Help file for use with a component.
Registering Components (see page 1352)	Registration works on a compilation unit basis, so if you create several components in a single compilation unit, you can register them all at once. To register a component, add a *Register* procedure to the unit. Within the *Register* procedure, you register the components and determine where to install them on the Tool palette. **Note:** If you create your component by choosing Component->New Component in the IDE, the code required to register your component is added automatically. The steps for manually registering a component are: • Declaring the Register procedure (see page 1347) • Writing the Register procedure (see page 1357)
Registering Multiple Properties at Once (see page 1352)	Register multiple properties at one time and associate them with a property category using the *RegisterPropertiesInCategory* function. *RegisterPropertiesInCategory* comes in three overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with property categories. The first variation lets you identify properties based on property name or type. The list is passed as an array of constants. In the example below, any property that either has the name "Text" or belongs to a class of type *TEdit* is registered in the category 'Localizable.'
Registering One Property at a Time (see page 1353)	Register one property at a time and associate it with a property category using the *RegisterPropertyInCategory* function. *RegisterPropertyInCategory* comes in four overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with the property category. The first variation lets you identify the property by the property's name. The line below registers a property related to visual display of the component, identifying the property by its name, "AutoSize".
Registering the Component Editor (see page 1353)	Once the component editor is defined, it can be registered to work with a particular component class. A registered component editor is created for each component of that class when it is selected in the form designer. To create the association between a component editor and a component class, call *RegisterComponentEditor*. *RegisterComponentEditor* takes the name of the component class that uses the editor, and the name of the component editor class that you have defined. For example, the following statement registers a component editor class named *TMyEditor* to work with all components of type *TMyComponent*:
Registering the Property Editor (see page 1354)	Once you create a property editor, you need to register it with Delphi. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component. To register a property editor, call the *RegisterPropertyEditor* procedure. *RegisterPropertyEditor* takes four parameters: • A type-information pointer for the type of property to edit—this is always a call to the built-in function *TypeInfo*, such as `TypeInfo(TMyComponent)__typeinfo(TMyComponent)`. • The type of the component to which this editor applies—if this... more (see page 1354)
Setting the Property Value (see page 1354)	The property editor's *SetValue* method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, *SetValue* should throw an exception and not use the improper value. To read string values into properties, override the property editor's *SetValue* method. *SetValue* should convert the string and validate the value before calling one of the Set methods.

Specifying Editor Attributes (see page 1354)	The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values. To specify editor attributes, override the property editor's *GetAttributes* method. *GetAttributes* is a method that returns a set of values of type *TPropertyAttributes* that can include any or all of the following values: ***Property-editor attribute flags***
Specifying Menu Items (see page 1355)	Override the *GetVerbCount* method to return the number of commands you are adding to the context menu. Override the *GetVerb* method to return the strings that should be added for each of these commands. When overriding *GetVerb*, add an ampersand (&) to a string to cause the following character to appear underlined in the context menu and act as a shortcut key for selecting the menu item. Be sure to add an ellipsis (...) to the end of a command if it brings up a dialog. *GetVerb* has a single parameter that indicates the index of the command. The... more (see page 1355)
Specifying the Components (see page 1355)	Within the Register procedure, pass the component names in an open array, which you can construct inside the call to RegisterComponents.
Specifying the Palette Page (see page 1356)	The palette category name is a string. If the name you give for the palette category does not already exist, Delphi creates a new category with that name. Delphi stores the names of the standard categories in string-list resources so that international versions of the product can name the categories in their native languages. If you want to install a component on one of the standard categories, you should obtain the string for the category name by calling the *LoadStr* function, passing the constant representing the string resource for that category, such as *srSystem* for the System category.
Troubleshooting Custom Components (C++) (see page 1356)	A common problem when registering and installing custom components is that the component does not appear in the list of components after the package is successfully installed. The most common causes for components not appearing in the list or on the palette: • Missing **PACKAGE** modifier on the `Register` function • Missing **PACKAGE** modifier on the class • Missing `#pragma package(smart_init)` in the C++ source file • `Register` function is not found in a namespace with the same name as the source code module name. • `Register` is not being successfully exported. Use `tdump` on the `.BPL` to look for the exported function: `tdump -ebpl... more (see page 1356)`
Using the IsPropertyInCategory Function (see page 1356)	An application can query the existing registered properties to determine whether a given property is already registered in a specified category. This can be especially useful in situations like a localization utility that checks the categorization of properties preparatory to performing its localization operations. Two overloaded variations of the *IsPropertyInCategory* function are available, allowing for different criteria in determining whether a property is in a category. The first variation lets you base the comparison criteria on a combination of the class type of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return *True... more (see page 1356)*
Using the RegisterComponents Function (see page 1357)	Within the *Register* procedure, call *RegisterComponents* to register the components in the classes array. *RegisterComponents* is a function that takes two parameters: the name of a Tool palette category and the array of component classes. Set the Page parameter to the name of the category on the Tool palette where the components should appear. If the named category already exists, the components are added to that category. If the named category does not exist, Delphi creates a new palette category with that name. Call RegisterComponents from the implementation of the Register procedure in one of the units that defines the... more (see page 1357)

Writing the Register Procedure (see page 1357)	Inside the *Register* procedure of a unit containing components, you must register each component you want to add to the Tool palette. If the unit contains several components, you can register them at the same time. To register a component, call the *RegisterComponents* procedure once for each category of the Tool palette to which you want to add components. *RegisterComponents* involves three important things: 1. Specifying the components. (see page 1355) 2. Specifying the palette page. (see page 1356) 3. Using the RegisterComponents function (see page 1357).

3.2.1.10.1 Adding Clipboard Formats

By default, when a user chooses Copy while a component is selected in the IDE, the component is copied in Delphi's internal format. It can then be pasted into another form or data module. Your component can copy additional formats to the Clipboard by overriding the *Copy* method.

For example, the following *Copy* method allows a *TImage* component to copy its picture to the Clipboard. This picture is ignored by the Delphi IDE, but can be pasted into other applications.

```
procedure TMyComponent.Copy;
var
  MyFormat : Word;
  AData,APalette : THandle;
begin
  TImage(Component).Picture.Bitmap.SaveToClipBoardFormat(MyFormat, AData, APalette);
  ClipBoard.SetAsHandle(MyFormat, AData);
end;
void __fastcall TMyComponentEditor::Copy(void)
{
WORD AFormat;
int AData;
HPALETTE APalette;
((TImage *)Component)->Picture->SaveToClipboardFormat(AFormat, AData, APalette);
TClipboard *pClip = Clipboard(); // don't clear the clipboard!
pClip->SetAsHandle(AFormat, AData);
}
```

3.2.1.10.2 Adding Component Editors

Component editors determine what happens when the component is double-clicked in the designer and add commands to the context menu that appears when the component is right-clicked. They can also copy your component to the Windows clipboard in custom formats.

If you do not give your components a component editor, Delphi uses the default component editor. The default component editor is implemented by the class *TDefaultEditor*. *TDefaultEditor* does not add any new items to a component's context menu. When the component is double-clicked, *TDefaultEditor* searches the properties of the component and generates (or navigates to) the first event handler it finds.

To add items to the context menu, change the behavior when the component is double-clicked, or add new clipboard formats, derive a new class from *TComponentEditor* and register its use with your component. In your overridden methods, you can use the *Component* property of *TComponentEditor* to access the component that is being edited.

Adding a custom component editor consists of the steps:

- Adding items to the context menu (see page 1345)
- Changing the double-click behavior (see page 1345)
- Adding clipboard formats (see page 1344)

- Registering the component editor (see page 1353)

3.2.1.10.3 Adding Items to the Context Menu

When the user right-clicks the component, the *GetVerbCount* and *GetVerb* methods of the component editor are called to build context menu. You can override these methods to add commands (verbs) to the context menu.

Adding items to the context menu requires the steps:

- Specifying menu items (see page 1355)
- Implementing commands (see page 1349)

3.2.1.10.4 Adding Property Editors

The Object Inspector provides default editing for all types of properties. You can, however, provide an alternate editor for specific properties by writing and registering property editors. You can register property editors that apply only to the properties in the components you write, but you can also create editors that apply to all properties of a certain type.

At the simplest level, a property editor can operate in either or both of two ways: displaying and allowing the user to edit the current value as a text string, and displaying a dialog box that permits some other kind of editing. Depending on the property being edited, you might find it useful to provide either or both kinds.

Writing a property editor requires these five steps:

1. Deriving a property-editor class (see page 1348).
2. Editing the property as text.
3. Editing the property as a whole (see page 1349).
4. Specifying editor attributes (see page 1354).
5. Registering the property editor (see page 1354).

3.2.1.10.5 Changing the Double-click Behavior

When the component is double-clicked, the *Edit* method of the component editor is called. By default, the *Edit* method executes the first command added to the context menu. Thus, in the previous example (see page 1349), double-clicking the component executes the *DoThis* command.

While executing the first command is usually a good idea, you may want to change this default behavior. For example, you can provide an alternate behavior if

- you are not adding any commands to the context menu.
- you want to display a dialog that combines several commands when the component is double-clicked.

Override the *Edit* method to specify a new behavior when the component is double-clicked. For example, the following *Edit* method brings up a font dialog when the user double-clicks the component:

```
procedure TMyEditor.Edit;
var
  FontDlg: TFontDialog;
begin
  FontDlg := TFontDialog.Create(Application);
  try
    if FontDlg.Execute then
      MyComponent.FFont.Assign(FontDlg.Font);
  finally
    FontDlg.Free
  end;
```

```
end;
void __fastcall TMyEditor::Edit(void)
{
TFontDialog *pFontDlg = new TFontDialog(NULL);
pFontDlg->Execute();
((TMyComponent *)Component)->Font = pFontDlg->Font;
delete pFontDlg;
}
```

Note: If you want a double-click on the component to display the Code editor for an event handler, use *TDefaultEditor* as a base class for your component editor instead of *TComponentEditor*. Then, instead of overriding the *Edit* method, override the protected *TDefaultEditor.EditProperty* method instead. *EditProperty* scans through the event handlers of the component, and brings up the first one it finds. You can change this to look a particular event instead. For example:

```
procedure TMyEditor.EditProperty(PropertyEditor: TPropertyEditor;
  Continue, FreeEditor: Boolean)
begin
  if (PropertyEditor.ClassName = 'TMethodProperty') and
    (PropertyEditor.GetName = 'OnSpecialEvent') then
    // DefaultEditor.EditProperty(PropertyEditor, Continue, FreeEditor);
end;
void __fastcall TMyEditor::EditProperty(TPropertyEditor* PropertyEditor,
bool &Continue, bool &FreeEditor)
{
if (PropertyEditor->ClassNameIs("TMethodProperty") &&
CompareText(PropertyEditor->GetName, "OnSpecialEvent") == 0)
{
TDefaultEditor::EditProperty(PropertyEditor, Continue, FreeEditor);
}
}
```

3.2.1.10.6 Compiling Components into Packages

Once your components are registered, you must compile them as packages before they can be installed in the IDE. A package can contain one or several components as well as custom property editors. For more information about packages, see Working with packages and components (see page 2211) .

To create and compile a package, see Creating and editing packages (see page 2213) . Put the source-code units for your custom components in the package's Contains list. If your components depend on other packages, include those packages in the Requires list.

To install your components in the IDE, see Installing component packages (see page 2217).

3.2.1.10.7 Creating the Entries

To make your component's Help integrate seamlessly with the Help for the rest of the components in the library, observe the following conventions:

Each component should have a Help topic:

The component topic should show which unit the component is declared in and briefly describe the component. The component topic should link to secondary windows that describe the component's position in the object hierarchy and list all of its properties, events, and methods. Application developers access this topic by selecting the component on a form and pressing F1. For an example of a component topic, place any component on a form and press F1.

The component topic must have a # footnote with a value unique to the topic. The # footnote uniquely identifies each topic by the Help system.

The component topic should have a K footnote for keyword searching in the Help system Index that includes the name of the

component class. For example, the keyword footnote for the *TMemo* component is "TMemo."

The component topic should also have a $ footnote that provides the title of the topic. The title appears in the Topics Found dialog box, the Bookmark dialog box, and the History window.

Each component should include the following secondary navigational topics:

- A hierarchy topic with links to every ancestor of the component in the component hierarchy.
- A list of all properties available in the component, with links to entries describing those properties.
- A list of all events available in the component, with links to entries describing those events.
- A list of methods available in the component, with links to entries describing those methods.

Links to object classes, properties, methods, or events in the Delphi Help system can be made using Alinks. When linking to an object class, the Alink uses the class name of the object, followed by an underscore and the string "object". For example, to link to the *TCustomPanel* object, use the following:

```
!AL(TCustomPanel_object,1)
```

When linking to a property, method, or event, precede the name of the property, method, or event by the name of the object that implements it and an underscore. For example, to link to the *Text* property which is implemented by *TControl*, use the following:

```
!AL(TControl_Text,1)
```

To see an example of the secondary navigation topics, display the Help for any component and click on the links labeled hierarchy, properties, methods, or events.

Each property, method, and event that is declared within the component should have a topic:

A property, event, or method topic should show the declaration of the item and describe its use. Application developers see these topics either by highlighting the item in the Object Inspector and pressing `F1` or by placing the cursor in the Code editor on the name of the item and pressing `F1`. To see an example of a property topic, select any item in the Object Inspector and press `F1`.

The property, event, and method topics should include a K footnote that lists the name of the property, method, or event, and its name in combination with the name of the component. Thus, the *Text* property of *TControl* has the following K footnote:

```
Text,TControl;TControl,Text;Text,
```

The property, method, and event topics should also include a $ footnote that indicates the title of the topic, such as TControl.Text.

All of these topics should have a topic ID that is unique to the topic, entered as a # footnote.

3.2.1.10.8 Creating the Help File

You can use any tool you want to create the source file for a Windows Help file (in .rtf format). Delphi includes the Microsoft Help Workshop, which compiles your Help files and provides an online Help authoring guide. You can find complete information about creating Help files in the online guide for Help Workshop.

Composing Help files for components consists of the steps:

- Creating the entries (see page 1346).
- Making component Help context-sensitive (see page 1350).

3.2.1.10.9 Declaring the Register Procedure

Registration involves writing a single procedure in the unit, which must have the name *Register*. The *Register* procedure must appear in the interface part of the unit, and (unlike the rest of Delphi) its name is case-sensitive.

Note: Although Delphi is a case insensitive language, the Register procedure is case sensitive and must be spelled with an

uppercase R.

The following code shows the outline of a simple unit that creates and registers new components:

```
unit MyBtns;
interface
type
  ...                                                { declare your component types here }
procedure Register;                                { this must appear in the interface section }
implementation
  ...                                                { component implementation goes here }
procedure Register;
begin
  ...                                                                { register the components }
end;
end.
namespace Newcomp
{
void __fastcall PACKAGE Register()
{
}
}
```

Within the *Register* procedure, call *RegisterComponents* for each component you want to add to the Tool palette. If the unit contains several components, you can register them all in one step.

3.2.1.10.10 Deriving a Property-editor Class

Both the component library define several kinds of property editors, all of which descend from *TPropertyEditor*. When you create a property editor, your property-editor class can either descend directly from *TPropertyEditor* or indirectly through one of the property-editor classes described in the table below. The classes in the DesignEditors unit can be used for VCL applications.

Note: All that is absolutely necessary for a property editor is that it descend from *TBasePropertyEditor* and that it support the *IProperty* interface. *TPropertyEditor*, however, provides a default implementation of the *IProperty* interface.

The list in the table below is not complete. The VCLEditors unit also defines some very specialized property editors used by unique properties such as the component name. The listed property editors are the ones that are the most useful for user-defined properties.

Predefined property-editor types

Type	Properties edited
TOrdinalProperty	All ordinal-property editors (those for integer, character, and enumerated properties) descend from *TOrdinalProperty*.
TIntegerProperty	All integer types, including predefined and user-defined subranges.
TCharProperty	*Char*-type and subranges of *Char*, such as 'A'..'Z'.
TEnumProperty	Any enumerated type.
TFloatProperty	All floating-point numbers.
TStringProperty	Strings.
TSetElementProperty	Individual elements in sets, shown as Boolean values
TSetProperty	All sets. Sets are not directly editable, but can expand into a list of set-element properties.
TClassProperty	Classes. Displays the name of the class and allows expansion of the class's properties.
TMethodProperty	Method pointers, most notably events.
TComponentProperty	Components in the same form. The user cannot edit the component's properties, but can point to a specific component of a compatible type.

TColorProperty	Component colors. Shows color constants if applicable, otherwise displays hexadecimal value. Drop down list contains the color constants. Double-click opens the color-selection dialog box.
TFontNameProperty	Font names. The drop down list displays all currently installed fonts.
TFontProperty	Fonts. Allows expansion of individual font properties as well as access to the font dialog box.

3.2.1.10.11 Editing the Property as a Whole

You can optionally provide a dialog box in which the user can visually edit a property. The most common use of property editors is for properties that are themselves classes. An example is the *Font* property, for which the user can open a font dialog box to choose all the attributes of the font at once.

To provide a whole-property editor dialog box, override the property-editor class's *Edit* method.

Edit methods use the same Get and Set methods used in writing *GetValue* and *SetValue* methods. In fact, an *Edit* method calls both a Get method and a Set method. Because the editor is type-specific, there is usually no need to convert the property values to strings. The editor generally deals with the value "as retrieved."

When the user clicks the '...' button next to the property or double-clicks the value column, the Object Inspector calls the property editor's *Edit* method.

Within your implementation of the Edit method, follow these steps:

1. Construct the editor you are using for the property.
2. Read the current value and assign it to the property using a Get method.
3. When the user selects a new value, assign that value to the property using a Set method.
4. Destroy the editor.

3.2.1.10.12 Implementing Commands

When the command provided by *GetVerb* is selected in the designer, the *ExecuteVerb* method is called. For every command you provide in the *GetVerb* method, implement an action in the *ExecuteVerb* method. You can access the component that is being edited using the *Component* property of the editor.

For example, the following *ExecuteVerb* method implements the commands for the *GetVerb* method in the previous (see page 1355) example.

```
procedure TMyEditor.ExecuteVerb(Index: Integer);
var
  MySpecialDialog: TMyDialog;
begin
  case Index of
    0: begin
         MyDialog := TMySpecialDialog.Create(Application);       { instantiate the editor }
         if MySpecialDialog.Execute then;                 { if the user OKs the dialog... }
           MyComponent.FThisProperty := MySpecialDialog.ReturnValue;   { ...use the value }
         MySpecialDialog.Free;                                    { destroy the editor }
       end;
    1: That;                                                 { call the That method }
  end;
end;
void __fastcall TMyEditor::ExecuteVerb(int Index)
{
switch (Index)
{
case 0:
TMyDialog *MySpecialDialog = new TMyDialog();
```

```
MySpecialDialog->Execute();
((TMyComponent *)Component)->ThisProperty = MySpecialDialog->ReturnValue;
delete MySpecialDialog;
break;
case 1:
That();  // call the "That" method
break;
}
}
```

3.2.1.10.13 Making Component Help Context-sensitive

Each component, property, method, and event topic must have an A footnote. The A footnote is used to display the topic when the user selects a component and presses F1, or when a property or event is selected in the Object Inspector and the user presses F1. The A footnotes must follow certain naming conventions:

If the Help topic is for a component, the A footnote consists of two entries separated by a semicolon using this syntax:

`ComponentClass_Object;ComponentClass`

where *ComponentClass* is the name of the component class.

If the Help topic is for a property or event, the A footnote consists of three entries separated by semicolons using this syntax:

`ComponentClass_Element;Element_Type;Element`

where *ComponentClass* is the name of the component class, *Element* is the name of the property, method, or event, and *Type* is the either Property, Method, or Event

For example, for a property named *BackgroundColor* of a component named *TMyGrid*, the A footnote is

`TMyGrid_BackgroundColor;BackgroundColor_Property;BackgroundColor`

3.2.1.10.14 Making Components Available at Design Time: Overview

Making your components available at design time requires several steps:

- Registering components (see page 1352)
- Providing Help for your component (see page 1351)
- Adding property editors (see page 1345)
- Adding component editors (see page 1344)
- Compiling components into packages (see page 1346)

Not all these steps apply to every component. For example, if you don't define any new properties or events, you don't need to provide Help for them. The only steps that are always necessary are registration and compilation.

Once your components have been registered and compiled into packages, they can be distributed to other developers and installed in the IDE. For information on installing packages in the IDE, see Installing component packages (see page 2217).

See Also

Extending the IDE (see page 1285)

3.2.1.10.15 Property Categories

In the IDE, the Object Inspector lets you selectively hide and display properties based on property categories. The properties of new custom components can be fit into this scheme by registering properties in categories. Do this at the same time you register the component by calling *RegisterPropertyInCategory* or *RegisterPropertiesInCategory*. Use *RegisterPropertyInCategory* to register a single property. Use *RegisterPropertiesInCategory* to register multiple properties in a single function call. These functions are defined in the unit DesignIntf.

Note that it is not mandatory that you register properties or that you register all of the properties of a custom component when some are registered. Any property not explicitly associated with a category is included in the *TMiscellaneousCategory* category. Such properties are displayed or hidden in the Object Inspector based on that default categorization.

In addition to these two functions for registering properties, there is an *IsPropertyInCategory* function. This function is useful for creating localization utilities, in which you must determine whether a property is registered in a given property category.

- Registering one property at a time (see page 1353)
- Registering multiple properties at once (see page 1352)
- Specifying property categories (see page 1351)
- Using the IsPropertyInCategory function (see page 1356)

3.2.1.10.16 Specifying Property Categories

When you register properties in a category, you can use any string you want as the name of the category. If you use a string that has not been used before, the Object Inspector generates a new property category class with that name. You can also, however, register properties into one of the categories that are built-in. The built-in property categories are described in the following table:

Property categories

Category	Purpose
Action	Properties related to runtime actions; the *Enabled* and *Hint* properties of *TEdit* are in this category.
Database	Properties related to database operations; the *DatabaseName* and *SQL* properties of *TQuery* are in this category.
Drag, Drop, and Docking	Properties related to drag-and-drop and docking operations; the *DragCursor* and *DragKind* properties of *TImage* are in this category.
Help and Hints	Properties related to using online Help or hints; the *HelpContext* and *Hint* properties of *TMemo* are in this category.
Layout	Properties related to the visual display of a control at design-time; the *Top* and *Left* properties of *TDBEdit* are in this category.
Legacy	Properties related to obsolete operations; the *Ctl3D* and *ParentCtl3D* properties of *TComboBox* are in this category.
Linkage	Properties related to associating or linking one component to another; the *DataSet* property of *TDataSource* is in this category.
Locale	Properties related to international locales; the *BiDiMode* and *ParentBiDiMode* properties of *TMainMenu* are in this category.
Localizable	Properties that may require modification in localized versions of an application. Many string properties (such as *Caption*) are in this category, as are properties that determine the size and position of controls.
Visual	Properties related to the visual display of a control at runtime; the *Align* and *Visible* properties of *TScrollBox* are in this category.
Input	Properties related to the input of data (need not be related to database operations); the *Enabled* and *ReadOnly* properties of *TEdit* are in this category.
Miscellaneous	Properties that do not fit a category or do not need to be categorized (and properties not explicitly registered to a specific category); the AllowAllUp and Name properties of TSpeedButton are in this category.

3.2.1.10.17 Providing Help for Your Component

When you select a standard component on a form, or a property or event in the Object Inspector, you can press F1 to get Help

on that item. You can provide developers with the same kind of documentation for your components if you create the appropriate Help files.

You can provide a small Help file to describe your components, and your Help file becomes part of the user's overall Delphi Help system.

See the section Creating the Help file (see page 1347) for information on how to compose the Help file for use with a component.

3.2.1.10.18 Registering Components

Registration works on a compilation unit basis, so if you create several components in a single compilation unit, you can register them all at once.

To register a component, add a *Register* procedure to the unit. Within the *Register* procedure, you register the components and determine where to install them on the Tool palette.

Note: If you create your component by choosing Component->New Component

in the IDE, the code required to register your component is added automatically. The steps for manually registering a component are:

- Declaring the Register procedure (see page 1347)
- Writing the Register procedure (see page 1357)

3.2.1.10.19 Registering Multiple Properties at Once

Register multiple properties at one time and associate them with a property category using the *RegisterPropertiesInCategory* function. *RegisterPropertiesInCategory* comes in three overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with property categories.

The first variation lets you identify properties based on property name or type. The list is passed as an array of constants. In the example below, any property that either has the name "Text" or belongs to a class of type *TEdit* is registered in the category 'Localizable.'

```
RegisterPropertiesInCategory('Localizable', ['Text', TEdit]);
RegisterPropertiesInCategory("Localizable", ARRAYOFCONST("Text", __typeinfo(TEdit)));
```

The second variation lets you limit the registered properties to those that belong to a specific component. The list of properties to register include only names, not types. For example, the following code registers a number of properties into the 'Help and Hints' category for all components:

```
RegisterPropertiesInCategory('Help and Hints', TComponent, ['HelpContext', 'Hint',
'ParentShowHint', 'ShowHint']);
RegisterPropertyInCategory("Help and Hints", __classid(TComponent),
ARRAYOFCONST("HelpContext", "Hint", "ParentShowHint"));
```

The third variation lets you limit the registered properties to those that have a specific type. As with the second variation, the list of properties to register can include only names:

```
RegisterPropertiesInCategory('Localizable', TypeInfo(String), ['Text', 'Caption']);
RegisterPropertiesInCategory("Localizable", __typeinfo(TStrings), ARRAYOFCONST("Lines",
"Commands"));
```

See the section Specifying property categories (see page 1351) for a list of the available property categories and a brief description of their uses.

3.2.1.10.20 Registering One Property at a Time

Register one property at a time and associate it with a property category using the *RegisterPropertyInCategory* function. *RegisterPropertyInCategory* comes in four overloaded variations, each providing a different set of criteria for identifying the property in the custom component to be associated with the property category.

The first variation lets you identify the property by the property's name. The line below registers a property related to visual display of the component, identifying the property by its name, "AutoSize".

```
RegisterPropertyInCategory('Visual', 'AutoSize');
RegisterPropertyInCategory("Visual", "AutoSize");
```

The second variation is much like the first, except that it limits the category to only those properties of the given name that appear on components of a given type. The example below registers (into the 'Help and Hints' category) a property named "HelpContext" of a component of the custom class *TMyButton*.

```
RegisterPropertyInCategory('Help and Hints', TMyButton, 'HelpContext');
RegisterPropertyInCategory("Help and Hints", __classid(TMyButton), "HelpContext");
```

The third variation identifies the property using its type rather than its name. The example below registers a property based on its type, Integer.

```
RegisterPropertyInCategory('Visual', TypeInfo(Integer));
RegisterPropertyInCategory("Visual", typeid(TArrangement));
```

The final variation uses both the property's type and its name to identify the property. The example below registers a property based on a combination of its type, *TBitmap*, and its name, "Pattern."

```
RegisterPropertyInCategory('Visual', TypeInfo(TBitmap), 'Pattern');
RegisterPropertyInCategory("Visual", typeid(TBitmap), "Pattern");
```

See the section Specifying property categories (see page 1351) for a list of the available property categories and a brief description of their uses.

3.2.1.10.21 Registering the Component Editor

Once the component editor is defined, it can be registered to work with a particular component class. A registered component editor is created for each component of that class when it is selected in the form designer.

To create the association between a component editor and a component class, call *RegisterComponentEditor*. *RegisterComponentEditor* takes the name of the component class that uses the editor, and the name of the component editor class that you have defined. For example, the following statement registers a component editor class named *TMyEditor* to work with all components of type *TMyComponent*:

```
RegisterComponentEditor(TMyComponent, TMyEditor);
RegisterComponentEditor(__classid( TMyComponent), __classid(TMyEditor));
```

Place the call to *RegisterComponentEditor* in the *Register* procedure where you register your component. For example, if a new component named *TMyComponent* and its component editor *TMyEditor* are both implemented in the same unit, the following code registers the component and its association with the component editor.

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TMyComponent);
  RegisterComponentEditor(classes[0], TMyEditor);
end;
namespace Newcomp
{
  void __fastcall PACKAGE Register()
  {
```

```
    RegisterComponents("Miscellaneous", classes, 0);
    RegisterComponentEditor(classes[0], __classid(TMyEditor));
  }
}
```

3.2.1.10.22 Registering the Property Editor

Once you create a property editor, you need to register it with Delphi. Registering a property editor associates a type of property with a specific property editor. You can register the editor with all properties of a given type or just with a particular property of a particular type of component.

To register a property editor, call the *RegisterPropertyEditor* procedure.

RegisterPropertyEditor takes four parameters:

- A type-information pointer for the type of property to edit—this is always a call to the built-in function *TypeInfo*, such as `TypeInfo(TMyComponent)`__`typeinfo(TMyComponent)`.
- The type of the component to which this editor applies—if this parameter is **nil**, the editor applies to all properties of the given type.
- The name of the property—this parameter only has meaning if the previous parameter specifies a particular type of component. In that case, you can specify the name of a particular property in that component type to which this editor applies.
- The type of property editor to use for editing the specified property.

3.2.1.10.23 Setting the Property Value

The property editor's *SetValue* method takes a string typed by the user in the Object Inspector, converts it into the appropriate type, and sets the value of the property. If the string does not represent a proper value for the property, *SetValue* should throw an exception and not use the improper value.

To read string values into properties, override the property editor's *SetValue* method.

SetValue should convert the string and validate the value before calling one of the Set methods.

3.2.1.10.24 Specifying Editor Attributes

The property editor must provide information that the Object Inspector can use to determine what tools to display. For example, the Object Inspector needs to know whether the property has subproperties or can display a list of possible values.

To specify editor attributes, override the property editor's *GetAttributes* method.

GetAttributes is a method that returns a set of values of type *TPropertyAttributes* that can include any or all of the following values:

Property-editor attribute flags

Flag	Related method	Meaning if included
paValueList	GetValues	The editor can give a list of enumerated values.
paSubProperties	GetProperties	The property has subproperties that can display.
paDialog	Edit	The editor can display a dialog box for editing the entire property.
paMultiSelect	N/A	The property should display when the user selects more than one component.
paAutoUpdate	SetValue	Updates the component after every change instead of waiting for approval of the value.

paSortList	N/A	The Object Inspector should sort the value list.
paReadOnly	N/A	Users cannot modify the property value.
paRevertable	N/A	Enables the Revert to Inherited menu item on the Object Inspector's context menu. The menu item tells the property editor to discard the current property value and return to some previously established default or standard value.
paFullWidthName	N/A	The value does not need to be displayed. The Object Inspector uses its full width for the property name instead.
paVolatileSubProperties	GetProperties	The Object Inspector re-fetches the values of all subproperties any time the property value changes.
paReference	GetComponentValue	The value is a reference to something else. When used in conjunction with paSubProperties the referenced object should be displayed as sub properties to this property.

3.2.1.10.25 Specifying Menu Items

Override the *GetVerbCount* method to return the number of commands you are adding to the context menu. Override the *GetVerb* method to return the strings that should be added for each of these commands. When overriding *GetVerb*, add an ampersand (&) to a string to cause the following character to appear underlined in the context menu and act as a shortcut key for selecting the menu item. Be sure to add an ellipsis (...) to the end of a command if it brings up a dialog. *GetVerb* has a single parameter that indicates the index of the command.

The following code overrides the *GetVerbCount* and *GetVerb* methods to add two commands to the context menu.

```
function TMyEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;
function TMyEditor.GetVerb(Index: Integer): String;
begin
  case Index of
    0: Result := '&DoThis ...';
    1: Result := 'Do&That';
  end;
end;
int __fastcall TMyEditor::GetVerbCount(void)
{
return 2;
}
System::AnsiStringBase __fastcall TMyEditor::GetVerb(int Index)
{
switch (Index)
{
case 0: return "&DoThis ..."; break;
case 1: return "Do&That"; break;
}
}
```

Note: Be sure that your *GetVerb* method returns a value for every possible index indicated by *GetVerbCount*.

3.2.1.10.26 Specifying the Components

Within the Register procedure, pass the component names in an open array, which you can construct inside the call to RegisterComponents.

```
RegisterComponents('Miscellaneous', [TMyComponent]);
TMetaClass classes[1] = {__classid(TNewComponent)};
```

You could also register several components on the same page at once, or register components on different pages, as shown in the following code:

```
procedure Register;
begin
  RegisterComponents('Miscellaneous', [TFirst, TSecond]);            { two on this page... }
  RegisterComponents('Assorted', [TThird]);                         { ...one on another... }
  RegisterComponents(LoadStr(srStandard), [TFourth]);    { ...and one on the Standard page }
end;
TMetaClass classes[2] =
{__classid(TNewComponent), __classid(TAnotherComponent)};
//Another way to add a component to the array
TMetaClass classes[2];
classes[0] = __classid(TNewComponent);
classes[1] = __classid(TAnotherComponent);
```

3.2.1.10.27 Specifying the Palette Page

The palette category name is a string. If the name you give for the palette category does not already exist, Delphi creates a new category with that name. Delphi stores the names of the standard categories in string-list resources so that international versions of the product can name the categories in their native languages. If you want to install a component on one of the standard categories, you should obtain the string for the category name by calling the *LoadStr* function, passing the constant representing the string resource for that category, such as *srSystem* for the System category.

3.2.1.10.28 Troubleshooting Custom Components (C++)

A common problem when registering and installing custom components is that the component does not appear in the list of components after the package is successfully installed.

The most common causes for components not appearing in the list or on the palette:

- Missing **PACKAGE** modifier on the `Register` function
- Missing **PACKAGE** modifier on the class
- Missing `#pragma package(smart_init)` in the C++ source file
- `Register` function is not found in a namespace with the same name as the source code module name.
- `Register` is not being successfully exported. Use `tdump` on the `.BPL` to look for the exported function:

```
tdump -ebpl mypack.bpl mypack.dmp
```

In the exports section of the dump, you should see the `Register` function (within the namespace) being exported.

3.2.1.10.29 Using the IsPropertyInCategory Function

An application can query the existing registered properties to determine whether a given property is already registered in a specified category. This can be especially useful in situations like a localization utility that checks the categorization of properties preparatory to performing its localization operations. Two overloaded variations of the *IsPropertyInCategory* function are available, allowing for different criteria in determining whether a property is in a category.

The first variation lets you base the comparison criteria on a combination of the class type of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return *True*, the property must belong to a *TCustomEdit* descendant, have the name "Text," and be in the property category 'Localizable'.

```
IsItThere := IsPropertyInCategory('Localizable', TCustomEdit, 'Text');
IsItThere = IsPropertyInCategory("Localizable", __classid(TCustomEdit), "Text");
```

The second variation lets you base the comparison criteria on a combination of the class name of the owning component and the property's name. In the command line below, for *IsPropertyInCategory* to return *True*, the property must be a *TCustomEdit*

descendant, have the name "Text", and be in the property category 'Localizable'.

```
IsItThere := IsPropertyInCategory('Localizable', 'TCustomEdit', 'Text');
IsItThere = IsPropertyInCategory("Localizable", "TCustomEdit", "Text");
```

3.2.1.10.30 Using the RegisterComponents Function

Within the *Register* procedure, call *RegisterComponents* to register the components in the classes array. *RegisterComponents* is a function that takes two parameters: the name of a Tool palette category and the array of component classes.

Set the Page parameter to the name of the category on the Tool palette where the components should appear. If the named category already exists, the components are added to that category. If the named category does not exist, Delphi creates a new palette category with that name.

Call RegisterComponents from the implementation of the Register procedure in one of the units that defines the custom components. The units that define the components must then be compiled into a package and the package must be installed before the custom components are added to the Tool palette.

```
procedure Register;
begin
  RegisterComponents('System', [TSystem1, TSystem2]);                    {add to system
category}
  RegisterComponents('MyCustomPage',[TCustom1, TCustom2]);                          { new
category}
end;
namespace Newcomp
{
void __fastcall PACKAGE Register()
{
TMetaClass* classes[1] = {__classid(TMyComponent)};
RegisterComponents("Miscellaneous", classes, 0);
}
}
namespace Mycomps
{
 void __fastcall PACKAGE Register()
{
// declares an array that holds two components
TMetaClass classes1[2] = {__classid(TFirst), __classid(TSecond)};
// adds a new palette page with the two components in the classes1 array
RegisterComponents("Miscellaneous", classes1, 1);
// declares a second array
TMetaClass classes2[1];
// assigns a component to be the first element in the array
classes2[0]  = __classid(TThird);
// adds the component in the classes2 array to the Samples page
RegisterComponents("Samples", classes2, 0);
}
}
```

3.2.1.10.31 Writing the Register Procedure

Inside the *Register* procedure of a unit containing components, you must register each component you want to add to the Tool palette. If the unit contains several components, you can register them at the same time.

To register a component, call the *RegisterComponents* procedure once for each category of the Tool palette to which you want to add components. *RegisterComponents* involves three important things:

1. Specifying the components. (see page 1355)
2. Specifying the palette page. (see page 1356)
3. Using the RegisterComponents function (see page 1357).

3.2.1.11 Making a dialog box a component

Topics

Name	Description
Adding Interface Properties (see page 1359)	Before proceeding, decide on the properties your wrapper needs to enable developers to use your dialog box as a component in their applications. Then, you can add declarations for those properties to the component's class declaration. Properties in wrapper components are somewhat simpler than the properties you would create if you were writing a regular component. Remember that in this case, you are just creating some persistent data that the wrapper can pass back and forth to the dialog box. By putting that data in the form of properties, you enable developers to set data at design time so that... more (see page 1359)
Adding the Execute Method (see page 1359)	The final part of the component interface is a way to open the dialog box and return a result when it closes. As with the common dialog box components, you use a boolean function called *Execute* that returns *True* if the user clicks OK, or *False* if the user cancels the dialog box. The declaration for the *Execute* method always looks like this:
Creating and Registering the Component (Dialog Box) (see page 1360)	Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. This process is outlined in Creating a new component. (see page 1317) For this example, follow the general procedure for creating a component, with these specifics: • Call the component's unit *AboutDlg*. • Derive a new component type called *TAboutBoxDlg*, descended from *TComponent*. • Register *TAboutBoxDlg* on the Samples page of the Tool palette. The resulting unit should look like this:
Creating the Component Interface (see page 1361)	These are the steps to create the component interface: 1. Including the form unit files (see page 1361). 2. Adding interface properties (see page 1359). 3. Adding the Execute method (see page 1359).
Defining the Component Interface (see page 1361)	Before you can create the component for your dialog box, you need to decide how you want developers to use it. You create an interface between your dialog box and applications that use it. For example, look at the properties for the common dialog box components. They enable the developer to set the initial state of the dialog box, such as the caption and initial control settings, then read back any needed information after the dialog box closes. There is no direct interaction with the individual controls in the dialog box, just with the properties in the wrapper component. The... more (see page 1361)
Including the Form Unit (see page 1361)	For your wrapper component to initialize and display the wrapped dialog box, you must add the form's unit to the **uses** clause of the wrapper component's unit. Append *About* to the **uses** clause of the *AboutDlg* unit. The **uses** clause now looks like this:

Making a Dialog Box a Component: Overview (see page 1362)	You will find it convenient to make a frequently used dialog box into a component that you add to the Tool palette. Your dialog box components will work just like the components that represent the standard common dialog boxes. The goal is to create a simple component that a user can add to a project and set properties for at design time. Making a dialog box a component requires these steps: 1. Defining the component interface (see page 1361) 2. Creating and registering the component (see page 1360) 3. Creating the component interface (see page 1361) 4. Testing the component (see page 1362) The Delphi "wrapper" component associated with the dialog box creates and executes the... more (see page 1362)
Testing the Component (see page 1362)	Once you have installed the dialog box component, you can use it as you would any of the common dialog boxes, by placing one on a form and executing it. A quick way to test the About box is to add a command button to a form and execute the dialog box when the user clicks the button.

3.2.1.11.1 Adding Interface Properties

Before proceeding, decide on the properties your wrapper needs to enable developers to use your dialog box as a component in their applications. Then, you can add declarations for those properties to the component's class declaration.

Properties in wrapper components are somewhat simpler than the properties you would create if you were writing a regular component. Remember that in this case, you are just creating some persistent data that the wrapper can pass back and forth to the dialog box. By putting that data in the form of properties, you enable developers to set data at design time so that the wrapper can pass it to the dialog box at runtime.

Declaring an interface property requires two additions to the component's class declaration:

- A private class field, which is a variable the wrapper uses to store the value of the property
- The published property declaration itself, which specifies the name of the property and tells it which field to use for storage

Interface properties of this sort do not need access methods. They use direct access to their stored data. By convention, the class field that stores the property's value has the same name as the property, but with the letter *F* in front. The field and the property *must* be of the same type.

3.2.1.11.2 Adding the Execute Method

The final part of the component interface is a way to open the dialog box and return a result when it closes. As with the common dialog box components, you use a boolean function called *Execute* that returns *True* if the user clicks OK, or *False* if the user cancels the dialog box.

The declaration for the *Execute* method always looks like this:

```
type
  TMyWrapper = class(TComponent)
  public
    function Execute: Boolean;
  end;
class PACKAGE TMyWrapper : public TComponent
{
  .
  .
  .
public:
    bool __fastcall Execute();
  .
```

```
  .
  .
};
```

The minimum implementation for *Execute* needs to construct the dialog box form, show it as a modal dialog box, and return either *True* or *False*, depending on the return value from *ShowModal*.

3.2.1.11.3 Creating and Registering the Component (Dialog Box)

Creation of every component begins the same way: create a unit, derive a component class, register it, compile it, and install it on the Tool palette. This process is outlined in Creating a new component. (see page 1317)

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *AboutDlg*.
- Derive a new component type called *TAboutBoxDlg*, descended from *TComponent*.
- Register *TAboutBoxDlg* on the Samples page of the Tool palette.

The resulting unit should look like this:

```
unit AboutDlg;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms;
type
  TAboutBoxDlg = class(TComponent)
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TAboutBoxDlg]);
end;
end.
#include <vcl\vcl.h>
#pragma hdrstop
#include "AboutDlg.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
static inline TAboutBoxDlg *ValidCtrCheck()
{
  return new TAboutBoxDlg(NULL);
}
//---------------------------------------------------------------------------
namespace AboutDlg {
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TAboutBoxDlg)};
    RegisterComponents("Samples", classes, 0);
  }
}
#ifndef AboutDlgH
#define AboutDlgH
//---------------------------------------------------------------------------
#include <vcl\sysutils.hpp>
#include <vcl\controls.hpp>
#include <vcl\classes.hpp>
#include <vcl\forms.hpp>
//---------------------------------------------------------------------------
class PACKAGE TAboutBoxDlg : public TComponent
{
private:
```

```
protected:
public:
__published:
};
//---------------------------------------------------------------------------
#endif
```

The new component now has only the capabilities built into *TComponent*. It is the simplest nonvisual component. In the next section (see page 1361), you will create the interface between the component and the dialog box.

3.2.1.11.4 Creating the Component Interface

These are the steps to create the component interface:

1. Including the form unit files (see page 1361).
2. Adding interface properties (see page 1359).
3. Adding the Execute method (see page 1359).

3.2.1.11.5 Defining the Component Interface

Before you can create the component for your dialog box, you need to decide how you want developers to use it. You create an interface between your dialog box and applications that use it.

For example, look at the properties for the common dialog box components. They enable the developer to set the initial state of the dialog box, such as the caption and initial control settings, then read back any needed information after the dialog box closes. There is no direct interaction with the individual controls in the dialog box, just with the properties in the wrapper component.

The interface must therefore contain enough information that the dialog box form can appear in the way the developer specifies and return any information the application needs. You can think of the properties in the wrapper component as being persistent data for a transient dialog box.

In the case of the About box, you do not need to return any information, so the wrapper's properties only have to contain the information needed to display the About box properly. Because there are four separate fields in the About box that the application might affect, you will provide four string-type properties to provide for them.

3.2.1.11.6 Including the Form Unit

For your wrapper component to initialize and display the wrapped dialog box, you must add the form's unit to the **uses** clause of the wrapper component's unit.

Append *About* to the **uses** clause of the *AboutDlg* unit.

The **uses** clause now looks like this:

```
uses
  Windows, SysUtils, Messages, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
Forms,
  About;
// for C++
#include "About.h"
#pragma link "About.obj"
```

The form unit always declares an instance of the form class. In the case of the About box, the form class is *TAboutBox*, and the *About* unit includes the following declaration:

```
var
  AboutBox: TAboutBox;
extern TAboutBox *AboutBox;
```

So by adding *About* to the **uses** clause, you make *AboutBox* available to the wrapper component.

3.2.1.11.7 Making a Dialog Box a Component: Overview

You will find it convenient to make a frequently used dialog box into a component that you add to the Tool palette. Your dialog box components will work just like the components that represent the standard common dialog boxes. The goal is to create a simple component that a user can add to a project and set properties for at design time.

Making a dialog box a component requires these steps:

1. Defining the component interface (see page 1361)
2. Creating and registering the component (see page 1360)
3. Creating the component interface (see page 1361)
4. Testing the component (see page 1362)

The Delphi "wrapper" component associated with the dialog box creates and executes the dialog box at runtime, passing along the data the user specified. The dialog-box component is therefore both reusable and customizable.

In this section, you will see how to create a wrapper component around the generic About Box form provided in the Delphi Object Repository.

Note: Copy the files ABOUT.PAS and ABOUT.DFM into your working directory.

There are not many special considerations for designing a dialog box that will be wrapped into a component. Nearly any form can operate as a dialog box in this context.

3.2.1.11.8 Testing the Component

Once you have installed the dialog box component, you can use it as you would any of the common dialog boxes, by placing one on a form and executing it. A quick way to test the About box is to add a command button to a form and execute the dialog box when the user clicks the button.

For example, if you created an About dialog box, made it a component, and added it to the Tool palette, you can test it with the following steps:

1. Create a new project.
2. Place an About box component on the main form.
3. Place a command button on the form.
4. Double-click the command button to create an empty click-event handler.
5. In the click-event handler, type the following line of code:

```
AboutBoxDlg1.Execute;
AboutBoxDlg1->Execute();
```

6. Run the application.

When the main form appears, click the command button. The About box appears with the default project icon and the name Project1. Choose OK to close the dialog box.

You can further test the component by setting the various properties of the About box component and again running the application.

3.2.1.12 Modifying an existing component

Topics

Name	Description
Creating and Registering the Modified Component (see page 1363)	You create every component the same way: you create a unit, derive a component class, register it, and install it on the Tool palette. This process is outlined in Creating a new component. (see page 1317) For this example, follow the general procedure for creating a component, with these specifics: • Call the component's unit *Memos*. • Derive a new component type called *TWrapMemo*, descended from *TMemo*. • Register *TWrapMemo* on the Samples page of the Tool palette. • The resulting unit should look like this:
Modifying an Existing Component: Overview (see page 1365)	The easiest way to create a component is to derive it from a component that does nearly everything you want, then make whatever changes you need. What follows is a simple example that modifies the standard memo component to create a memo that does not wrap words by default. The value of the memo component's *WordWrap* property is initialized to *True*. If you frequently use non-wrapping memos, you can create a new memo component that does not wrap words by default. **Note:** To modify published properties or save specific event handlers for an existing component, it is often easier... more (see page 1365)
Modifying the Component Object (see page 1365)	Once you have created a new component class, you can modify it in almost any way. In this case, you will change only the initial value of one property in the memo component. This involves two small changes to the component class: • Overriding the constructor. • Specifying the new default property value. The constructor actually sets the value of the property. The default tells Delphi what values to store in the form (.dfm for VCL applications) file. Delphi stores only values that differ from the default, so it is important to perform both steps.
Overriding the Constructor (see page 1365)	When a component is placed on a form at design time, or when an application constructs a component at runtime, the component's constructor sets the property values. When a component is loaded from a form file, the application sets any properties changed at design time. **Note:** When you override a constructor, the new constructor must call the inherited constructor before doing anything else. For more information, see Overriding methods (see page 1374). For this example, your new component needs to override the constructor inherited from *TMemo* to set the *WordWrap* property to *False*. To achieve this, add the constructor override to... more (see page 1365)
Specifying the New Default Property Value (see page 1366)	When Delphi stores a description of a form in a form file, it stores the values only of properties that differ from their defaults. Storing only the differing values keeps the form files small and makes loading the form faster. If you create a property or change the default value, it is a good idea to update the property declaration to include the new default. Form files, loading, and default values are explained in more detail in Making components available at design time (see page 1350). To change the default value of a property, redeclare the property name, followed by the directive... more (see page 1366)

3.2.1.12.1 Creating and Registering the Modified Component

You create every component the same way: you create a unit, derive a component class, register it, and install it on the Tool palette. This process is outlined in Creating a new component. (see page 1317)

For this example, follow the general procedure for creating a component, with these specifics:

- Call the component's unit *Memos*.
- Derive a new component type called *TWrapMemo*, descended from *TMemo*.
- Register *TWrapMemo* on the Samples page of the Tool palette.
- The resulting unit should look like this:

```
unit Memos;
interface
uses
 SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
 Forms, StdCtrls;
type
 TWrapMemo = class(TMemo)
 end;
procedure Register;
implementation
procedure Register;
begin
 RegisterComponents('Samples', [TWrapMemo]);
end;
end.
#include <vcl.h>
#pragma hdrstop
#include "Yelmemo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init);
//---------------------------------------------------------------------------
// ValidCtrCheck is used to assure that the components created do not have
// any pure virtual functions.
//
static inline void ValidCtrCheck(TYellowMemo *)
{
  new TYellowMemo(NULL);
}
//---------------------------------------------------------------------------
__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
: TMemo(Owner)
{
}
//---------------------------------------------------------------------------
namespace Yelmemo
{
  void __fastcall PACKAGE Register()
  {
    TComponentClass classes[1] = {__classid(TYellowMemo)};
    RegisterComponents("Samples", classes, 0); //"Common Controls" in CLX applications
  }
}
#ifndef YelMemoH
#define YelmemoH
//---------------------------------------------------------------------------
#include <sysutils.hpp>
#include <controls.hpp>
#include <classes.hpp>
#include <forms.hpp>
#include <StdCtrls.hpp>
//---------------------------------------------------------------------------
class PACKAGE TYellowMemo : public TMemo
{
private:
protected:
public:
__published:
};
```

```
//---------------------------------------------------------------------------
#endif
```

If you compile and install the new component now, it behaves exactly like its ancestor, *TMemo*. In the next section, you will make a simple change to your component.

3.2.1.12.2 Modifying an Existing Component: Overview

The easiest way to create a component is to derive it from a component that does nearly everything you want, then make whatever changes you need. What follows is a simple example that modifies the standard memo component to create a memo that does not wrap words by default.

The value of the memo component's *WordWrap* property is initialized to *True*. If you frequently use non-wrapping memos, you can create a new memo component that does not wrap words by default.

Note: To modify published properties or save specific event handlers for an existing component, it is often easier to use a *component template* rather than create a new class.

Modifying an existing component takes only two steps:

- Creating and registering the component (see page 1363).
- Modifying the component class (see page 1365).

3.2.1.12.3 Modifying the Component Object

Once you have created a new component class, you can modify it in almost any way. In this case, you will change only the initial value of one property in the memo component. This involves two small changes to the component class:

- Overriding the constructor.
- Specifying the new default property value.

The constructor actually sets the value of the property. The default tells Delphi what values to store in the form (.dfm for VCL applications) file. Delphi stores only values that differ from the default, so it is important to perform both steps.

3.2.1.12.4 Overriding the Constructor

When a component is placed on a form at design time, or when an application constructs a component at runtime, the component's constructor sets the property values. When a component is loaded from a form file, the application sets any properties changed at design time.

Note: When you override a constructor, the new constructor must call the inherited constructor before doing anything else. For more information, see Overriding methods (see page 1374).

For this example, your new component needs to override the constructor inherited from *TMemo* to set the *WordWrap* property to *False*. To achieve this, add the constructor override to the forward declaration, then write the new constructor in the **implementation** part of the unit:

```
type
  TWrapMemo = class(TMemo)
  public                                              { constructors are always public }
    constructor Create(AOwner: TComponent); override; { this syntax is always the same }
  end;
.
.
.
constructor TWrapMemo.Create(AOwner: TComponent);     { this goes after implementation }
begin
  inherited Create(AOwner);                           { ALWAYS do this first! }
```

```
  WordWrap := False;                                    { set the new desired value }
end;
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner); // the constructor declaration
__published:
    __property Color;
};
__fastcall TYellowMemo::TYellowMemo(TComponent* Owner)
  : TMemo(Owner)                                     // the constructor implementation
first...                                                    // ...calls the constructor for TMemo
{
  Color = clYellow;                                 // colors the component yellow
}
```

Now you can install the new component on the Tool palette and add it to a form. Note that the *WordWrap* property is now initialized to *False*.

If you change an initial property value, you should also designate that value as the default. If you fail to match the value set by the constructor to the specified default value, Delphi cannot store and restore the proper value.

3.2.1.12.5 Specifying the New Default Property Value

When Delphi stores a description of a form in a form file, it stores the values only of properties that differ from their defaults. Storing only the differing values keeps the form files small and makes loading the form faster. If you create a property or change the default value, it is a good idea to update the property declaration to include the new default. Form files, loading, and default values are explained in more detail in Making components available at design time (see page 1350).

To change the default value of a property, redeclare the property name, followed by the directive **default** and the new default value. You don't need to redeclare the entire property, just the name and the default value.

For the word-wrapping memo component, you redeclare the *WordWrap* property in the **published** part of the object declaration, with a default value of *False*:

```
type
  TWrapMemo = class(TMemo)
  .
  .
  .
  published
    property WordWrap default False;
  end;
//header file
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
__published:
    __property Color = {default=clYellow};
};
//implmentation file
__fastcall TYellowMemo::TYellowMemo(TComponent* AOwner) : TMemo(AOwner)
{
  Color = clYellow;
  WordWrap = false;
}
//header file with WordWrap as default value of false:
class PACKAGE TYellowMemo : public TMemo
{
public:
    virtual __fastcall TYellowMemo(TComponent* Owner);
__published:
```

```
    __property Color = {default=clYellow};
    __property WordWrap = {default=false};
};
```

Specifying the default property value does not affect the workings of the component. You must still initialize the value in the component's constructor. Redeclaring the default ensures that Delphi knows when to write *WordWrap* to the form file.

3.2.1.13 Object-oriented programming for component writers

Topics

Name	Description
Object-oriented Programming for Component Writers: Overview (see page 1369)	If you have written applications with Delphi, you know that a class contains both data and code, and that you can manipulate classes at design time and at runtime. In that sense, you've become a component user. When you create new components, you deal with classes in ways that application developers never need to. You also try to hide the inner workings of the component from the developers who will use it. By choosing appropriate ancestors for your components, designing interfaces that expose only the properties and methods that developers need, and following the other guidelines in the following topics,... more (see page 1369)
Defining New Classes (see page 1370)	The difference between component writers and application developers is that component writers create new classes while application developers manipulate instances of classes. A class is essentially a type. As a programmer, you are always working with types and instances, even if you do not use that terminology. For example, you create variables of a type, such as *Integer*. Classes are usually more complex than simple data types, but they work the same way: By assigning different values to instances of the same type, you can perform different tasks. For example, it is quite common to create a form... more (see page 1370)
Deriving New Classes (see page 1370)	There are two reasons to derive a new class: • To change class defaults to avoid repetition (see page 1371) • To add new capabilities to a class (see page 1370) In either case, the goal is to create reusable objects. If you design components with reuse in mind, you can save work later on. Give your classes usable default values, but allow them to be customized.
Abstract Class Members (see page 1370)	When a method is declared as **abstract** in an ancestor class, you should surface it (by redeclaring and implementing it) in any descendant component before you use the new component in applications. On the Win32 platform, Delphi can create instances of a class that contains abstract members. This is not recommended, however. For more information about surfacing inherited parts of classes, see Creating properties (see page 1249) and Creating methods. (see page 1243)
Adding New Capabilities to a Class (see page 1370)	A common reason for creating new components is to add capabilities not found in existing components. When you do this, you derive the new component from either an existing component or an abstract base class, such as *TComponent* or *TControl*. Derive your new component from the class that contains the closest subset of the features you want. You can add capabilities to a class, but you cannot take them away; so if an existing component class contains properties that you do *not* want to include in yours, you should derive from that component's ancestor. For example, if you want... more (see page 1370)
Ancestors, Descendants, and Class Hierarchies (see page 1371)	Application developers take for granted that every control has properties named *Top* and *Left* that determine its position on the form. To them, it may not matter that all controls inherit these properties from a common ancestor, *TControl*. When you create a component, however, you must know which class to derive it from so that it inherits the appropriate features. And you must know everything that your control inherits, so you can take advantage of inherited features without recreating them. The class from which you derive a component is called its *immediate ancestor*. Each component inherits from its... more (see page 1371)

Changing Class Defaults to Avoid Repetition (see page 1371)	Most programmers try to avoid repetition. Thus, if you find yourself rewriting the same lines of code over and over, you place the code in a subroutine or function, or build a library of routines that you can use in many programs. The same reasoning holds for components. If you find yourself changing the same properties or making the same method calls, you can create a new component that does these things by default. For example, suppose that each time you create an application, you add a dialog box to perform a particular operation. Although it is not difficult to... more (see page 1371)
Controlling Access (see page 1371)	There are five levels of *access control* - also called *visibility* - on properties, methods, and fields. Visibility determines which code can access which parts of the class. By specifying visibility, you define the *interface* to your components. The table below shows the levels of visibility, from most restrictive to most accessible: ***Levels of visibility within an object***
Declaring a New Component Class (see page 1372)	In addition to standard components, Delphi provides many abstract classes designed as bases for deriving new components. The Creating components (see page 1315) topic shows the classes you can start from when you create your own components. To declare a new component class, add a class declaration to the component's unit file.
Defining the Design-time Interface (see page 1372)	Declaring part of a class as **published** makes that part public and also generates runtime type information. Among other things, runtime type information allows the Object Inspector to access properties and events. Because they show up in the Object Inspector, the published parts of a class define that class's *design-time interface*. The design-time interface should include any aspects of the class that an application developer might want to customize at design time, but must exclude any properties that depend on specific information about the runtime environment. Read-only properties cannot be part of the design-time interface because the application developer... more (see page 1372)
Defining the Component Writer's Interface (see page 1373)	Declaring part of a class as **protected** makes that part visible only to the class itself and its descendants (and to other classes that share their unit files). You can use **protected** declarations to define a *component writer's interface* to the class. Application units do not have access to the protected parts, but derived classes do. This means that component writers can change the way a class works without making the details visible to application developers. **Note:** A common mistake is trying to access protected methods from an event handler. Event handlers are typically methods of the form, not the... more (see page 1373)
Defining the Runtime Interface (see page 1373)	Declaring part of a class as **public** makes that part visible to any code that has access to the class as a whole. Public parts are available at runtime to all code, so the public parts of a class define its *runtime interface*. The runtime interface is useful for items that are not meaningful or appropriate at design time, such as properties that depend on runtime input or which are read-only. Methods that you intend for application developers to call must also be public.
Dispatching Methods (see page 1373)	*Dispatch* refers to the way a program determines where a method should be invoked when it encounters a method call. The code that calls a method looks like any other procedure or function call. But classes have different ways of dispatching methods. The three types of method dispatch are • Static • Virtual • Dynamic
Dynamic Methods (see page 1373)	Dynamic methods are virtual methods with a slightly different dispatch mechanism. Because dynamic methods don't have entries in the object's virtual method table, they can reduce the amount of memory that objects consume. However, dispatching dynamic methods is somewhat slower than dispatching regular virtual methods. If a method is called frequently, or if its execution is time-critical, you should probably declare it as virtual rather than dynamic. Objects must store the addresses of their dynamic methods. But instead of receiving entries in the virtual method table, dynamic methods are listed separately. The dynamic method list contains entries only for methods... more (see page 1373)
Hiding Implementation Details (see page 1374)	Declaring part of a class as **private** makes that part invisible to code outside the class's unit file. Within the unit that contains the declaration, code can access the part as if it were public.

Virtual Methods (see page 1374)	Virtual methods employ a more complicated, and more flexible, dispatch mechanism than static methods. A virtual method can be redefined in descendant classes, but still be called in the ancestor class. The address of a virtual method isn't determined at compile time; instead, the object where the method is defined looks up the address at runtime. To make a method virtual, add the directive **virtual** after the method declaration. The **virtual** directive creates an entry in the object's *virtual method table*, or VMT, which holds the addresses of all the virtual methods in an object type. When you derive... more (see page 1374)
Classes and Pointers (see page 1374)	Every class (and therefore every component) is really a pointer. The compiler automatically dereferences class pointers for you, so most of the time you do not need to think about this. The status of classes as pointers becomes important when you pass a class as a parameter. In general, you should pass classes by value rather than by reference. The reason is that classes are already pointers, which are references; passing a class by reference amounts to passing a reference to a reference.
Overriding Methods (see page 1374)	*Overriding* a method means extending or refining it, rather than replacing it. A descendant class can override any of its inherited virtual methods. To override a method in a descendant class, add the directive **override** to the end of the method declaration. Overriding a method causes a compilation error if • The method does not exist in the ancestor class. • The ancestor's method of that name is static. • The declarations are not otherwise identical (number and type of arguments parameters differ).
Regular Methods (C++) (see page 1375)	Class methods are regular (or nonvirtual) unless you specifically declare them as virtual, or unless they override a **virtual** method in a base class. The compiler can determine the exact address of a regular class member at compile time. This is known as compile-time binding. A base class regular method is inherited by derived classes. In the following example, an object of type `Derived` can call the method `Regular()` as it were its own method. Declaring a method in a derived class with the same name and parameters as a regular method in the class's ancestor replaces the ancestor's... more (see page 1375)
Static Methods (see page 1375)	All methods are static unless you specify otherwise when you declare them. Static methods work like regular procedures or functions. The compiler determines the exact address of the method and links the method at compile time. The primary advantage of static methods is that dispatching them is very quick. Because the compiler can determine the exact address of the method, it links the method directly. Virtual and dynamic methods, by contrast, use indirect means to look up the address of their methods at runtime, which takes somewhat longer. A static method does not change when inherited by a descendant class.... more (see page 1375)

3.2.1.13.1 Object-oriented Programming for Component Writers: Overview

If you have written applications with Delphi, you know that a class contains both data and code, and that you can manipulate classes at design time and at runtime. In that sense, you've become a component user.

When you create new components, you deal with classes in ways that application developers never need to. You also try to hide the inner workings of the component from the developers who will use it. By choosing appropriate ancestors for your components, designing interfaces that expose only the properties and methods that developers need, and following the other guidelines in the following topics, you can create versatile, reusable components.

Before you start creating components, you should be familiar with these topics, which are related to object-oriented programming (OOP):

- Defining new classes (see page 1370)
- Ancestors (see page 1371)
- Controlling access (see page 1371)
- Dispatching methods (see page 1373)

- Abstract class members (see page 1370)
- Classes and pointers (see page 1374)

3.2.1.13.2 Defining New Classes

The difference between component writers and application developers is that component writers create new classes while application developers manipulate instances of classes.

A class is essentially a type. As a programmer, you are always working with types and instances, even if you do not use that terminology. For example, you create variables of a type, such as *Integer*. Classes are usually more complex than simple data types, but they work the same way: By assigning different values to instances of the same type, you can perform different tasks.

For example, it is quite common to create a form containing two buttons, one labeled OK and one labeled Cancel. Each is an instance of the class *TButton*, but by assigning different values to their *Caption* properties and different handlers to their *OnClick* events, you make the two instances behave differently.

See Also

Deriving New Types (see page 1370)

Declaring a New Component Type (see page 1372)

3.2.1.13.3 Deriving New Classes

There are two reasons to derive a new class:

- To change class defaults to avoid repetition (see page 1371)
- To add new capabilities to a class (see page 1370)

In either case, the goal is to create reusable objects. If you design components with reuse in mind, you can save work later on. Give your classes usable default values, but allow them to be customized.

3.2.1.13.4 Abstract Class Members

When a method is declared as **abstract** in an ancestor class, you should surface it (by redeclaring and implementing it) in any descendant component before you use the new component in applications. On the Win32 platform, Delphi can create instances of a class that contains abstract members. This is not recommended, however. For more information about surfacing inherited parts of classes, see Creating properties (see page 1249) and Creating methods. (see page 1243)

3.2.1.13.5 Adding New Capabilities to a Class

A common reason for creating new components is to add capabilities not found in existing components. When you do this, you derive the new component from either an existing component or an abstract base class, such as *TComponent* or *TControl*.

Derive your new component from the class that contains the closest subset of the features you want. You can add capabilities to a class, but you cannot take them away; so if an existing component class contains properties that you do *not* want to include in yours, you should derive from that component's ancestor.

For example, if you want to add features to a list box, you could derive your component from *TListBox*. However, if you want to add new features but exclude some capabilities of the standard list box, you need to derive your component from *TCustomListBox*, the ancestor of *TListBox*. Then you can recreate (or make visible) only the list-box capabilities you want, and add your new features.

Customizing a grid (see page 1264) shows an example of customizing an abstract component class.

3.2.1.13.6 Ancestors, Descendants, and Class Hierarchies

Application developers take for granted that every control has properties named *Top* and *Left* that determine its position on the form. To them, it may not matter that all controls inherit these properties from a common ancestor, *TControl*. When you create a component, however, you must know which class to derive it from so that it inherits the appropriate features. And you must know everything that your control inherits, so you can take advantage of inherited features without recreating them.

The class from which you derive a component is called its *immediate ancestor*. Each component inherits from its immediate ancestor, and from the immediate ancestor of its immediate ancestor, and so forth. All of the classes from which a component inherits are called its *ancestors*; the component is a *descendant* of its ancestors.

Together, all the ancestor-descendant relationships in an application constitute a hierarchy of classes. Each generation in the hierarchy contains more than its ancestors, since a class inherits everything from its ancestors, then adds new properties and methods or redefines existing ones.

If you do not specify an immediate ancestor, Delphi derives your component from the default ancestor, *TObject*. *TObject* is the ultimate ancestor of all classes in the object hierarchy.

The general rule for choosing which object to derive from is simple: Pick the object that contains as much as possible of what you want to include in your new object, but which does not include anything you do not want in the new object. You can always add things to your objects, but you cannot take things out.

3.2.1.13.7 Changing Class Defaults to Avoid Repetition

Most programmers try to avoid repetition. Thus, if you find yourself rewriting the same lines of code over and over, you place the code in a subroutine or function, or build a library of routines that you can use in many programs. The same reasoning holds for components. If you find yourself changing the same properties or making the same method calls, you can create a new component that does these things by default.

For example, suppose that each time you create an application, you add a dialog box to perform a particular operation. Although it is not difficult to recreate the dialog each time, it is also not necessary. You can design the dialog once, set its properties, and install a wrapper component associated with it onto the Tool palette. By making the dialog into a reusable component, you not only eliminate a repetitive task, but you encourage standardization and reduce the likelihood of errors each time the dialog is recreated.

Modifying an existing component (see page 1365) shows an example of changing a component's default properties.

Note: If you want to modify only the published properties of an existing component, or to save specific event handlers for a component or group of components, you may be able to accomplish this more easily by creating a *component template*.

3.2.1.13.8 Controlling Access

There are five levels of *access control* - also called *visibility* - on properties, methods, and fields. Visibility determines which code can access which parts of the class. By specifying visibility, you define the *interface* to your components.

The table below shows the levels of visibility, from most restrictive to most accessible:

Levels of visibility within an object

Visibility	Meaning	Used for
private	Accessible only to code in the unit where the class is defined.	Hiding implementation details. (see page 1374)

protected	Accessible to code in the unit(s) where the class and its descendants are defined.	Defining the component writer's interface. (see page 1373)
public	Accessible to all code.	Defining the runtime interface. (see page 1373)
automated	Accessible to all code. Automation type information is generated.	OLE automation only.
published	Accessible to all code and accessible from the Object Inspector. Saved in a form file.	Defining the design-time interface. (see page 1372)

Declare members as **private** if you want them to be available only within the class where they are defined; declare them as **protected** if you want them to be available only within that class and its descendants. Remember, though, that if a member is available anywhere within a unit file, it is available *everywhere* in that file. Thus, if you define two classes in the same unit, the classes will be able to access each other's private methods. And if you derive a class in a different unit from its ancestor, all the classes in the new unit will be able to access the ancestor's protected methods.

See Also

Hiding Implementation Details (see page 1374)

Defining the Component Writer's Interface (see page 1373)

Defining the Runtime Interface (see page 1373)

Defining the Design-time Interface (see page 1372)

3.2.1.13.9 Declaring a New Component Class

In addition to standard components, Delphi provides many abstract classes designed as bases for deriving new components. The Creating components (see page 1315) topic shows the classes you can start from when you create your own components.

To declare a new component class, add a class declaration to the component's unit file.

See Also

Deriving New Types (see page 1370)

3.2.1.13.10 Defining the Design-time Interface

Declaring part of a class as **published** makes that part public and also generates runtime type information. Among other things, runtime type information allows the Object Inspector to access properties and events.

Because they show up in the Object Inspector, the published parts of a class define that class's *design-time interface*. The design-time interface should include any aspects of the class that an application developer might want to customize at design time, but must exclude any properties that depend on specific information about the runtime environment.

Read-only properties cannot be part of the design-time interface because the application developer cannot assign values to them directly. Read-only properties should therefore be public, rather than published.

See Also

Hiding Implementation Details (see page 1374)

Defining the Component Writer's Interface (see page 1373)

Defining the Runtime Interface (see page 1373)

3.2.1.13.11 Defining the Component Writer's Interface

Declaring part of a class as **protected** makes that part visible only to the class itself and its descendants (and to other classes that share their unit files).

You can use **protected** declarations to define a *component writer's interface* to the class. Application units do not have access to the protected parts, but derived classes do. This means that component writers can change the way a class works without making the details visible to application developers.

Note: A common mistake is trying to access protected methods from an event handler. Event handlers are typically methods of the form, not the component that receives the event. As a result, they do not have access to the component's protected methods (unless the component is declared in the same unit as the form).

See Also

Hiding Implementation Details (see page 1374)

Defining the Runtime Interface (see page 1373)

Defining the Design-time Interface (see page 1372)

3.2.1.13.12 Defining the Runtime Interface

Declaring part of a class as **public** makes that part visible to any code that has access to the class as a whole.

Public parts are available at runtime to all code, so the public parts of a class define its *runtime interface*. The runtime interface is useful for items that are not meaningful or appropriate at design time, such as properties that depend on runtime input or which are read-only. Methods that you intend for application developers to call must also be public.

See Also

Hiding Implementation Details (see page 1374)

Defining the Component Writer's Interface (see page 1373)

Defining the Design-time Interface (see page 1372)

3.2.1.13.13 Dispatching Methods

Dispatch refers to the way a program determines where a method should be invoked when it encounters a method call. The code that calls a method looks like any other procedure or function call. But classes have different ways of dispatching methods.

The three types of method dispatch are

- Static
- Virtual
- Dynamic

3.2.1.13.14 Dynamic Methods

Dynamic methods are virtual methods with a slightly different dispatch mechanism. Because dynamic methods don't have entries in the object's virtual method table, they can reduce the amount of memory that objects consume. However, dispatching dynamic methods is somewhat slower than dispatching regular virtual methods. If a method is called frequently, or if its execution is time-critical, you should probably declare it as virtual rather than dynamic.

Objects must store the addresses of their dynamic methods. But instead of receiving entries in the virtual method table, dynamic

methods are listed separately. The dynamic method list contains entries only for methods introduced or overridden by a particular class. (The virtual method table, in contrast, includes all of the object's virtual methods, both inherited and introduced.) Inherited dynamic methods are dispatched by searching each ancestor's dynamic method list, working backwards through the inheritance tree.

To make a method dynamic, add the directive **dynamic** after the method declaration.

3.2.1.13.15 Hiding Implementation Details

Declaring part of a class as **private** makes that part invisible to code outside the class's unit file. Within the unit that contains the declaration, code can access the part as if it were public.

See Also

Defining the Component Writer's Interface (see page 1373)

Defining the Runtime Interface (see page 1373)

Defining the Design-time Interface (see page 1372)

3.2.1.13.16 Virtual Methods

Virtual methods employ a more complicated, and more flexible, dispatch mechanism than static methods. A virtual method can be redefined in descendant classes, but still be called in the ancestor class. The address of a virtual method isn't determined at compile time; instead, the object where the method is defined looks up the address at runtime.

To make a method virtual, add the directive **virtual** after the method declaration. The **virtual** directive creates an entry in the object's *virtual method table*, or VMT, which holds the addresses of all the virtual methods in an object type.

When you derive a new class from an existing one, the new class gets its own VMT, which includes all the entries from the ancestor's VMT plus any additional virtual methods declared in the new class.

See Also

Regular Methods (see page 1375)

Overriding Methods (see page 1374)

3.2.1.13.17 Classes and Pointers

Every class (and therefore every component) is really a pointer. The compiler automatically dereferences class pointers for you, so most of the time you do not need to think about this. The status of classes as pointers becomes important when you pass a class as a parameter. In general, you should pass classes by value rather than by reference. The reason is that classes are already pointers, which are references; passing a class by reference amounts to passing a reference to a reference.

3.2.1.13.18 Overriding Methods

Overriding a method means extending or refining it, rather than replacing it. A descendant class can override any of its inherited virtual methods.

To override a method in a descendant class, add the directive **override** to the end of the method declaration.

Overriding a method causes a compilation error if

- The method does not exist in the ancestor class.
- The ancestor's method of that name is static.

- The declarations are not otherwise identical (number and type of arguments parameters differ).

See Also

Making Methods Virtual (see page 1245)

3.2.1.13.19 Regular Methods (C++)

Class methods are regular (or nonvirtual) unless you specifically declare them as virtual, or unless they override a **virtual** method in a base class. The compiler can determine the exact address of a regular class member at compile time. This is known as compile-time binding.

A base class regular method is inherited by derived classes. In the following example, an object of type `Derived` can call the method `Regular()` as it were its own method. Declaring a method in a derived class with the same name and parameters as a regular method in the class's ancestor replaces the ancestor's method. In the following example, when `d->AnotherRegular()` is called, it is being dispatched to the `Derived` class replacement for `AnotherRegular()`.

```
class Base
{
public:
    void Regular();
    void AnotherRegular();
    virtual void Virtual();
};
class Derived : public Base
{
public:
    void AnotherRegular();          // replaces Base::AnotherRegular()
    void Virtual();                 // overrides Base::Virtual()
};
void FunctionOne()
{
  Derived *d;
  d = new Derived;
  d->Regular();                   // Calling Regular() as it were a member of Derived
                                  // The same as calling d->Base::Regular()
  d->AnotherRegular();            // Calling the redefined AnotherRegular(), ...
                                  // ... the replacement for Base::AnotherRegular()
  delete d;
}
void FunctionTwo(Base *b)
{
  b->Virtual();
  b->AnotherRegular();
}
```

3.2.1.13.20 Static Methods

All methods are static unless you specify otherwise when you declare them. Static methods work like regular procedures or functions. The compiler determines the exact address of the method and links the method at compile time.

The primary advantage of static methods is that dispatching them is very quick. Because the compiler can determine the exact address of the method, it links the method directly. Virtual and dynamic methods, by contrast, use indirect means to look up the address of their methods at runtime, which takes somewhat longer.

A static method does not change when inherited by a descendant class. If you declare a class that includes a static method, then derive a new class from it, the derived class shares exactly the same method at the same address. This means that you cannot override static methods; a static method always does exactly the same thing no matter what class it is called in. If you declare a method in a derived class with the same name as a static method in the ancestor class, the new method simply replaces the inherited one in the derived class.

3.2.1.14 Using graphics in components

Topics

Name	Description
Creating and Managing Off-screen Bitmaps (see page 1378)	When creating complex graphic images, you should avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas. The most common use of an off-screen bitmap is in the *Paint* method of a graphic control. As with any temporary object, the bitmap should be protected with a **try**..**finally** block:
Loading and Storing Graphics (see page 1378)	All pictures and graphics in Delphi can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time. To load an image into a picture from a file, call the picture's *LoadFromFile* method. To save an image from a picture into a file, call the picture's *SaveToFile* method. *LoadFromFile* and *SaveToFile* each take the name of a file as the only parameter. *LoadFromFile* uses the extension of the file name to determine what kind of graphic object it will create and load. *SaveToFile* saves... more (see page 1378)
Handling Palettes (see page 1378)	For VCL components, when running on a palette-based device (typically, a 256-color video mode), Delphi controls automatically support palette realization. That is, if you have a control that has a palette, you can use two methods inherited from *TControl* to control how Windows accommodates that palette. Palette support for controls has these two aspects: • Specifying a palette for a control (see page 1380) • Responding to palette changes (see page 1380) Most controls have no need for a palette, but controls that contain "rich color" graphic images (such as the image control) might need to interact with Windows and the screen device driver to ensure the proper... more (see page 1378)
Off-screen Bitmaps (see page 1379)	When drawing complex graphic images, a common technique in graphics programming is to create an off-screen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an off-screen image reduces flicker caused by repeated drawing directly to the screen. The bitmap class in Delphi, which represents bitmapped images in resources and files, can also work as an off-screen image. There are two main aspects to working with off-screen bitmaps: • Creating and managing off-screen bitmaps. (see page 1378) • Copying bitmapped images.
Overview of Graphics (see page 1379)	Delphi encapsulates the Windows GDI at several levels. The most important to you as a component writer is the way components display their images on the screen. When calling GDI functions directly, you need to have a handle to a device context, into which you have selected various drawing tools such as pens, brushes, and fonts. After rendering your graphic images, you must restore the device context to its original state before disposing of it. Instead of forcing you to deal with graphics at a detailed level, Delphi provides a simple yet complete interface: your component's *Canvas* property. The canvas... more (see page 1379)

Using a Picture, Graphic, or Canvas (see page 1379)	There are three kinds of classes in Delphi that deal with graphics: • A *canvas* represents a bitmapped drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a stand-alone class. • A *graphic* represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or metafile. Delphi defines classes *TBitmap*, *TIcon*, and *TMetafile*, all descended from a generic *TGraphic*. You can also define your own graphic classes. By defining a minimal standard interface for all graphics, *TGraphic* provides a... more (see page 1379)
Responding to Changes (see page 1380)	All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images. Responding to changes in graphic objects is particularly important if you publish them as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects. To respond to... more (see page 1380)
Responding to Palette Changes (see page 1380)	If your VCL control specifies a palette by overriding *GetPalette*, Delphi automatically takes care of responding to palette messages from Windows. The method that handles the palette messages is *PaletteChanged*. The primary role of *PaletteChanged* is to determine whether to realize the control's palette in the foreground or the background. Windows handles this realization of palettes by making the topmost window have a foreground palette, with other windows resolved in background palettes. Delphi goes one step further, in that it also realizes palettes for controls within a window in tab order. The only time you might need to... more (see page 1380)
Specifying a Palette for a Control (see page 1380)	To specify a palette for a control, override the control's *GetPalette* method to return the handle of the palette. Specifying the palette for a control does these things for your application: • It tells the application that your control's palette needs to be realized. • It designates the palette to use for realization.
Using Graphics in Components: Overview (see page 1380)	Windows provides a powerful graphics device interface (GDI) for drawing device-independent graphics. The GDI, however, imposes extra requirements on the programmer, such as managing graphic resources. Delphi takes care of all the GDI drudgery, allowing you to focus on productive work instead of searching for lost handles or unreleased resources. As with any part of the Windows API, you can call GDI functions directly from your Delphi application. But you will probably find that using Delphi's encapsulation of the graphic functions is faster and easier. The topics in this section include: • Overview of graphics (see page 1379) • Using the canvas (see page 1381) • Working with pictures (see page 1381)... more (see page 1380)
Using the Canvas (see page 1381)	The canvas class encapsulates graphics controls at several levels, including high-level functions for drawing individual lines, shapes, and text; intermediate properties for manipulating the drawing capabilities of the canvas; and in the component library, provides low-level access to the Windows GDI. The following table summarizes the capabilities of the canvas. ***Canvas capability summary***

Working with Pictures (see page 1381)	Most of the graphics work you do in Delphi is limited to drawing directly on the canvases of components and forms. Delphi also provides for handling stand-alone graphic images, such as bitmaps, metafiles, and icons, including automatic management of palettes. There are three important aspects to working with pictures in Delphi: • Using a picture (see page 1379) • Loading and storing graphics (see page 1378) • Handling palettes (see page 1378)

3.2.1.14.1 Creating and Managing Off-screen Bitmaps

When creating complex graphic images, you should avoid drawing them directly on a canvas that appears onscreen. Instead of drawing on the canvas for a form or control, you can construct a bitmap object, draw on its canvas, and then copy its completed image to the onscreen canvas.

The most common use of an off-screen bitmap is in the *Paint* method of a graphic control. As with any temporary object, the bitmap should be protected with a **try**..**finally** block:

```
type
  TFancyControl = class(TGraphicControl)
  protected
    procedure Paint; override;                              { override the Paint method }
  end;
procedure TFancyControl.Paint;
var
  Bitmap: TBitmap;                          { temporary variable for the off-screen bitmap }
begin
  Bitmap := TBitmap.Create;                                { construct the bitmap object }
  try
    { draw on the bitmap }
    { copy the result into the control's canvas }
  finally
    Bitmap.Free;                                            { destroy the bitmap object }
  end;
end;
```

3.2.1.14.2 Loading and Storing Graphics

All pictures and graphics in Delphi can load their images from files and store them back again (or into different files). You can load or store the image of a picture at any time.

To load an image into a picture from a file, call the picture's *LoadFromFile* method. To save an image from a picture into a file, call the picture's *SaveToFile* method.

LoadFromFile and *SaveToFile* each take the name of a file as the only parameter. *LoadFromFile* uses the extension of the file name to determine what kind of graphic object it will create and load. *SaveToFile* saves whatever type of file is appropriate for the type of graphic object being saved.

3.2.1.14.3 Handling Palettes

For VCL components, when running on a palette-based device (typically, a 256-color video mode), Delphi controls automatically support palette realization. That is, if you have a control that has a palette, you can use two methods inherited from *TControl* to control how Windows accommodates that palette.

Palette support for controls has these two aspects:

- Specifying a palette for a control (see page 1380)

- Responding to palette changes (see page 1380)

Most controls have no need for a palette, but controls that contain "rich color" graphic images (such as the image control) might need to interact with Windows and the screen device driver to ensure the proper appearance of the control. Windows refers to this process as *realizing* palettes.

Realizing palettes is the process of ensuring that the foremost window uses its full palette, and that windows in the background use as much of their palettes as possible, then map any other colors to the closest available colors in the "real" palette. As windows move in front of one another, Windows continually realizes the palettes.

Note: Delphi itself provides no specific support for creating or maintaining palettes, other than in bitmaps. If you have a palette handle, however, Delphi controls can manage it for you.

3.2.1.14.4 Off-screen Bitmaps

When drawing complex graphic images, a common technique in graphics programming is to create an off-screen bitmap, draw the image on the bitmap, and then copy the complete image from the bitmap to the final destination onscreen. Using an off-screen image reduces flicker caused by repeated drawing directly to the screen.

The bitmap class in Delphi, which represents bitmapped images in resources and files, can also work as an off-screen image.

There are two main aspects to working with off-screen bitmaps:

- Creating and managing off-screen bitmaps. (see page 1378)
- Copying bitmapped images.

3.2.1.14.5 Overview of Graphics

Delphi encapsulates the Windows GDI at several levels. The most important to you as a component writer is the way components display their images on the screen. When calling GDI functions directly, you need to have a handle to a device context, into which you have selected various drawing tools such as pens, brushes, and fonts. After rendering your graphic images, you must restore the device context to its original state before disposing of it.

Instead of forcing you to deal with graphics at a detailed level, Delphi provides a simple yet complete interface: your component's *Canvas* property. The canvas ensures that it has a valid device context, and releases the context when you are not using it. Similarly, the canvas has its own properties representing the current pen, brush, and font.

The canvas manages all these resources for you, so you need not concern yourself with creating, selecting, and releasing things like pen handles. You just tell the canvas what kind of pen it should use, and it takes care of the rest.

One of the benefits of letting Delphi manage graphic resources is that it can cache resources for later use, which can speed up repetitive operations. For example, if you have a program that repeatedly creates, uses, and disposes of a particular kind of pen tool, you need to repeat those steps each time you use it. Because Delphi caches graphic resources, chances are good that a tool you use repeatedly is still in the cache, so instead of having to recreate a tool, Delphi uses an existing one.

An example of this is an application that has dozens of forms open, with hundreds of controls. Each of these controls might have one or more *TFont* properties. Though this could result in hundreds or thousands of instances of *TFont* objects, most applications wind up using only two or three font handles, thanks to a font cache.

3.2.1.14.6 Using a Picture, Graphic, or Canvas

There are three kinds of classes in Delphi that deal with graphics:

- A *canvas* represents a bitmapped drawing surface on a form, graphic control, printer, or bitmap. A canvas is always a property of something else, never a stand-alone class.
- A *graphic* represents a graphic image of the sort usually found in a file or resource, such as a bitmap, icon, or metafile. Delphi defines classes *TBitmap*, *TIcon*, and *TMetafile*, all descended from a generic *TGraphic*. You can also define your own graphic

classes. By defining a minimal standard interface for all graphics, *TGraphic* provides a simple mechanism for applications to use different kinds of graphics easily.

- A *picture* is a container for a graphic, meaning it could contain any of the graphic classes. That is, an item of type *TPicture* can contain a bitmap, an icon, a metafile, or a user-defined graphic type, and an application can access them all in the same way through the picture class. For example, the image control has a property called *Picture*, of type *TPicture*, enabling the control to display images from many kinds of graphics.

Keep in mind that a picture class always has a graphic, and a graphic might have a canvas. (The only standard graphic that has a canvas is *TBitmap*.) Normally, when dealing with a picture, you work only with the parts of the graphic class exposed through *TPicture*. If you need access to the specifics of the graphic class itself, you can refer to the picture's *Graphic* property.

3.2.1.14.7 Responding to Changes

All graphic objects, including canvases and their owned objects (pens, brushes, and fonts) have events built into them for responding to changes in the object. By using these events, you can make your components (or the applications that use them) respond to changes by redrawing their images.

Responding to changes in graphic objects is particularly important if you publish them as part of the design-time interface of your components. The only way to ensure that the design-time appearance of the component matches the properties set in the Object Inspector is to respond to changes in the objects.

To respond to changes in a graphic object, assign a method to the class's *OnChange* event.

3.2.1.14.8 Responding to Palette Changes

If your VCL control specifies a palette by overriding *GetPalette*, Delphi automatically takes care of responding to palette messages from Windows. The method that handles the palette messages is *PaletteChanged*.

The primary role of *PaletteChanged* is to determine whether to realize the control's palette in the foreground or the background. Windows handles this realization of palettes by making the topmost window have a foreground palette, with other windows resolved in background palettes. Delphi goes one step further, in that it also realizes palettes for controls within a window in tab order. The only time you might need to override this default behavior is if you want a control that is not first in tab order to have the foreground palette.

3.2.1.14.9 Specifying a Palette for a Control

To specify a palette for a control, override the control's *GetPalette* method to return the handle of the palette.

Specifying the palette for a control does these things for your application:

- It tells the application that your control's palette needs to be realized.
- It designates the palette to use for realization.

3.2.1.14.10 Using Graphics in Components: Overview

Windows provides a powerful graphics device interface (GDI) for drawing device-independent graphics. The GDI, however, imposes extra requirements on the programmer, such as managing graphic resources. Delphi takes care of all the GDI drudgery, allowing you to focus on productive work instead of searching for lost handles or unreleased resources.

As with any part of the Windows API, you can call GDI functions directly from your Delphi application. But you will probably find that using Delphi's encapsulation of the graphic functions is faster and easier.

The topics in this section include:

- Overview of graphics (see page 1379)

- Using the canvas (see page 1381)
- Working with pictures (see page 1381)
- Off-screen bitmaps (see page 1379)
- Responding to changes (see page 1380)

3.2.1.14.11 Using the Canvas

The canvas class encapsulates graphics controls at several levels, including high-level functions for drawing individual lines, shapes, and text; intermediate properties for manipulating the drawing capabilities of the canvas; and in the component library, provides low-level access to the Windows GDI.

The following table summarizes the capabilities of the canvas.

Canvas capability summary

Level	Operation	Tools
High	Drawing lines and shapes	Methods such as *MoveTo*, *LineTo*, *Rectangle*, and *Ellipse*
	Displaying and measuring text	*TextOut*, *TextHeight*, *TextWidth*, and *TextRect* methods
	Filling areas	*FillRect* and *FloodFill* methods
Intermediate	Customizing text and graphics	*Pen*, *Brush*, and *Font* properties
	Manipulating pixels	*Pixels* property.
	Copying and merging images	*Draw, StretchDraw, BrushCopy,* and *CopyRect* methods; *CopyMode* property
Low	Calling Windows GDI functions	*Handle* property

3.2.1.14.12 Working with Pictures

Most of the graphics work you do in Delphi is limited to drawing directly on the canvases of components and forms. Delphi also provides for handling stand-alone graphic images, such as bitmaps, metafiles, and icons, including automatic management of palettes.

There are three important aspects to working with pictures in Delphi:

- Using a picture (see page 1379)
- Loading and storing graphics (see page 1378)
- Handling palettes (see page 1378)

3.2.2 Developing COM-based Applications

Contains the Developer's Guide topics for creating COM-based applications in Delphi.

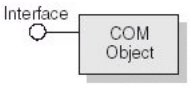
Topics

Name	Description
COM basics (see page 1382)	
Creating an Active Server Page (see page 1400)	
Using ActiveX controls (see page 1406)	
Creating COM clients (see page 1418)	

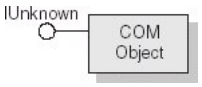
Creating simple COM servers (see page 1427)	
Working with type libraries (see page 1441)	

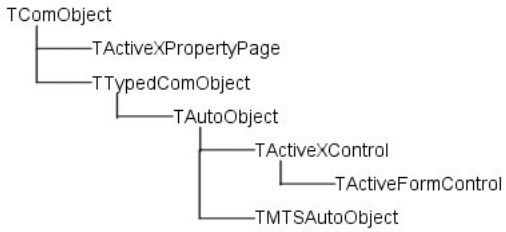
3.2.2.1 COM basics

Topics

Name	Description
Overview of COM Technologies (see page 1385)	Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms. COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly... more (see page 1385)
Parts of a COM Application (see page 1386)	When implementing a COM application, you supply the following:
COM Interfaces (see page 1386)	COM clients communicate with objects through COM interfaces. Interfaces are groups of logically or semantically related routines which provide communication between a provider of a service (server object) and its clients. The standard way to depict a COM interface is as follows:  For example, every COM object must implement the basic interface, IUnknown (see page 1387). Through a routine called *QueryInterface* in *IUnknown*, clients can request other interfaces implemented by the server. Objects can have multiple interfaces, where each interface implements a feature. An interface provides a way to convey to the client what service it provides, without... more (see page 1386)
The Fundamental COM Interface, IUnknown (see page 1387)	All COM objects must support the fundamental interface, called *IUnknown, a typedefto the base interface type IInterface. IUnknown contains the following routines:*
COM Interface Pointers (see page 1387)	An interface pointer is a pointer to an object instance that points, in turn, to the implementation of each method in the interface. The implementation is accessed through an array of pointers to these methods, which is called a **vtable**. Vtables are similar to the mechanism used to support virtual functions in Delphi. Because of this similarity, the compiler can resolve calls to methods on the interface the same way it resolves calls to methods on Delphi classes. The vtable is shared among all instances of an object class, so for each object instance, the object code allocates a... more (see page 1387)
COM Servers (see page 1388)	A COM server is an application or a library that provides services to a client application or library. A COM server consists of one or more COM objects, where a COM object is a set of properties and methods. Clients do not know *how* a COM object performs its service; the object's implementation remains encapsulated. An object makes its services available through its interfaces (see page 1386). In addition, clients do not need to know *where* a COM object resides. COM provides transparent access regardless of the object's location. (see page 1389) When a client requests a service from a COM object, the client passes... more (see page 1388)
CoClasses and Class Factories (see page 1388)	A COM object is an instance of a **CoClass**, which is a class that implements one or more COM interfaces. The COM object provides the services as defined by its interfaces. CoClasses are instantiated by a special type of object called a *class factory*. Whenever an object's services are requested by a client, a class factory creates an object instance for that particular client. Typically, if another client requests the object's services, the class factory creates another object instance to service the second client. (Clients can also bind to running COM objects that register themselves to support it.)... more (see page 1388)

In-process, Out-of-process, and Remote Servers (see page 1389)	With COM, a client does not need to know where an object resides, it simply makes a call to an object's interface. COM performs the necessary steps to make the call. These steps differ depending on whether the object resides in the same process as the client, in a different process on the client machine, or in a different machine across the network. The different types of servers are known as:
The Marshaling Mechanism (see page 1390)	Marshaling is the mechanism that allows a client to make interface function calls to remote objects in another process or on a different machine. Marshaling • Takes an interface pointer in the server's process and makes a proxy pointer available to code in the client process. • Transfers the arguments of an interface call as passed from the client and places the arguments into the remote object's process space. For any interface call, the client pushes arguments onto a stack and makes a function call through the interface pointer. If the call to the object is not in-process, the call gets passed... more (see page 1390)
Aggregation (COM) (see page 1391)	Sometimes, a server object makes use of another COM object to perform some of its functions. For example, an inventory management object might make use of a separate invoicing object to handle customer invoices. If the inventory management object wants to present the invoice interface to clients, however, there is a problem: Although a client that has the inventory interface can call *QueryInterface* to obtain the invoice interface, when the invoice object was created it did not know about the inventory management object and can't return an inventory interface in response to a call to *QueryInterface*. A client that... more (see page 1391)
COM Clients (see page 1391)	Clients can always query the interfaces of a COM object to determine what it is capable of providing. All COM objects allow clients to request known interfaces. In addition, if the server supports the *IDispatch* interface, clients can query the server for information about what methods the interface supports. Server objects have no expectations about the client using its objects. Similarly, clients don't need to know how (or even where) an object provides the services; they simply rely on server objects to provide the services they advertise through their interfaces. There are two types of COM clients, controllers and containers.... more (see page 1391)
COM Extensions (see page 1392)	COM was originally designed to provide core communication functionality and to enable the broadening of this functionality through extensions. COM itself has extended its core functionality by defining specialized sets of interfaces for specific purposes. The following lists some of the services COM extensions currently provide.
Automation Servers (see page 1393)	Automation refers to the ability of an application to control the objects in another application programmatically, like a macro that can manipulate more than one application at the same time. The server object being manipulated is called the Automation object, and the client of the Automation object is referred to as an Automation controller. Automation can be used on in-process, local, and remote servers. Automation is characterized by two key points: • The Automation object defines a set of properties and commands, and describes their capabilities through type descriptions. In order to do this, it must have a way to provide... more (see page 1393)
Active Server Pages (see page 1393)	The Active Server Page (ASP) technology lets you write simple scripts, called Active Server Pages, that can be launched by clients via a Web server. Unlike ActiveX controls, which run on the client, Active Server Pages run on the server, and return a resulting HTML page to clients. Active Server Pages are written in Jscript or VB script. The script runs every time the server loads the Web page. That script can then launch an embedded Automation server (or Enterprise Java Bean). For example, you can write an Automation server, such as one to create a bitmap or connect... more (see page 1393)

ActiveX Controls (see page 1394)	ActiveX is a technology that allows COM components, especially controls, to be more compact and efficient. This is especially necessary for controls that are intended for Intranet applications that need to be downloaded by a client before they are used. ActiveX controls are visual controls that run only as in-process servers, and can be plugged into an ActiveX control container application. They are not complete applications in themselves, but can be thought of as prefabricated OLE controls that are reusable in various applications. ActiveX controls have a visible user interface, and rely on predefined interfaces to negotiate I/O and display... more (see page 1394)
Active Documents (see page 1394)	Active Documents (previously referred to as OLE documents) are a set of COM services that support linking and embedding, drag-and-drop, and visual editing. Active Documents can seamlessly incorporate data or objects of different formats, such as sound clips, spreadsheets, text, and bitmaps. Unlike ActiveX controls, Active Documents are not limited to in-process servers; they can be used in cross-process applications. Unlike Automation objects, which are almost never visual, Active Document objects can be visually active in another application. Thus, Active Document objects are associated with two types of data: presentation data, used for visually displaying the object on a display... more (see page 1394)
Type Libraries (see page 1395)	Type libraries provide a way to get more type information about an object than can be determined from an object's interface. The type information contained in type libraries provides needed information about objects and their interfaces, such as what interfaces exist on what objects (given the CLSID), what member functions exist on each interface, and what arguments those functions require. You can obtain type information either by querying a running instance of an object or by loading and reading type libraries. With this information, you can implement a client which uses a desired object, knowing specifically what member functions you... more (see page 1395)
Implementing COM Objects with Wizards (see page 1397)	Delphi makes it easier to write COM server applications by providing wizards that handle many of the details involved. Delphi provides separate wizards to create the following: • A simple COM object • An Automation object • A COM+ Event Object • A Type library • An ActiveX library The wizards handle many of the tasks involved in creating each type of COM object. They provide the required COM interfaces for each type of object. With a simple COM object, the wizard implements the one required COM interface, *IUnknown*, which provides an interface pointer to the object.  The COM object... more (see page 1397)

Code Generated by Wizards (see page 1398)	Delphi's wizards generate classes that are derived from the Delphi ActiveX framework (DAX). Despite its name, the Delphi ActiveX framework supports all types of COM objects, not just ActiveX controls. The classes in this framework provide the underlying implementation of the standard COM interfaces for the objects you create using a wizard. The following figure illustrates the objects in the Delphi ActiveX framework:  Each wizard generates an implementation unit that implements your COM server object. The COM server object (the implementation object) descends from one of the classes in DAX: ***DAX Base classes for generated implementation classes... more (see page 1398)***
COM+ Event And Event Subscriber Objects (see page 1399)	The COM+ Events system introduces a middle layer of software that decouples applications that generate events (called publishers) from applications that respond to events (called subscribers). Instead of being tightly bound to each other, publishers and subscribers can be developed, deployed and activated independently of each other. In the COM+ Events model, an event interface is first created using the COM+ Event Object wizard. The event interface has no implementation; it simply defines the event methods that publishers will generate, and that subscribers will respond to. The COM+ event object is then installed into a COM+ Application, in the COM+... more (see page 1399)

3.2.2.1.1 Overview of COM Technologies

Delphi provides wizards and classes to make it easy to implement applications based on the Component Object Model (COM) from Microsoft. With these wizards, you can create COM-based classes and components to use within applications or you can create fully functional COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms.

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM). For more information on clients, servers, and interfaces see Parts of a COM Application (see page 1386).

COM as a specification and implementation

COM is both a specification and an implementation. The COM specification defines how objects are created and how they communicate with each other. According to this specification, COM objects can be written in different languages, run in different process spaces and on different platforms. As long as the objects adhere to the written specification, they can communicate. This allows you to integrate legacy code as a component with new components implemented in object-oriented languages.

The COM implementation is built into the Win32 subsystem, which provides a number of core services that support the written specification. The COM library contains a set of standard interfaces that define the core functionality of a COM object, and a small set of API functions designed for the purpose of creating and managing COM objects.

When you use Delphi wizards and VCL objects in your application, you are using Delphi's implementation of the COM specification. In addition, Delphi provides some wrappers for COM services for those features that it does not implement directly. You can find these wrappers defined in the `ComObj` unit and the API definitions in the `AxCtrls` unit.

Note: Delphi's interfaces and language follow the COM specification. Delphi implements objects conforming to the COM spec using a set of classes called the Delphi ActiveX framework (DAX). These classes are found in the `AxCtrls`, `OleCtrls`, and `OleServer` units. In addition, the Delphi interface to the COM API is in ActiveX.pas

and `ComSvcs.pas`.

COM extensions

As COM has evolved, it has been extended beyond the basic COM services. COM serves as the basis for other technologies such as Automation, ActiveX controls, and Active Directories. For details on COM extensions, see COM Extensions (see page 1392).

Delphi provides wizards to easily implement applications that incorporate the above technologies in the Delphi environment. For details, see Implementing COM Objects with Wizards (see page 1397).

See Also

Creating Simple COM Servers: Overview (see page 1430)

Creating Active Server Pages: Overview (see page 1400)

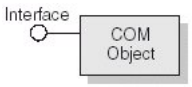
3.2.2.1.2 Parts of a COM Application

When implementing a COM application, you supply the following:

COM interface (see page 1386)	The way in which an object exposes its services externally to clients. A COM object provides an interface for each set of related methods and properties. Note that COM properties are not identical to properties on VCL objects. COM properties always use read and write access methods.
COM server (see page 1388)	A module, either an EXE, DLL, or OCX, that contains the code for a COM object. Object implementations reside in servers. A COM object implements one or more interfaces.
COM client (see page 1391)	The code that calls the interfaces to get the requested services from the server. Clients know what they want to get from the server (through the interface); clients do not know the internals of how the server provides the services. Delphi eases the process in creating a client by letting you install COM servers (such as a Word document or PowerPoint slide) as components on the **Tool Palette**. This allows you to connect to the server and hook its events through the **Object Inspector**.

3.2.2.1.3 COM Interfaces

COM clients communicate with objects through COM interfaces. Interfaces are groups of logically or semantically related routines which provide communication between a provider of a service (server object) and its clients. The standard way to depict a COM interface is as follows:



For example, every COM object must implement the basic interface, IUnknown (see page 1387). Through a routine called *QueryInterface* in *IUnknown*, clients can request other interfaces implemented by the server.

Objects can have multiple interfaces, where each interface implements a feature. An interface provides a way to convey to the client what service it provides, without providing implementation details of how or where the object provides this service.

Key aspects of COM interfaces are as follows:

- Once published, interfaces are immutable; that is, they do not change. You can rely on an interface to provide a specific set of functions. Additional functionality is provided by additional interfaces.

- By convention, COM interface identifiers begin with a capital I and a symbolic name that defines the interface, such as *IMalloc* or *IPersist*.
- Interfaces are guaranteed to have a unique identification, called a **Globally Unique Identifier (GUID)**, which is a 128-bit randomly generated number. Interface GUIDs are called **Interface Identifiers (IIDs)**. This eliminates naming conflicts between different versions of a product or different products.
- Interfaces are language independent. You can use any language to implement a COM interface as long as the language supports a structure of pointers, and can call a function through a pointer either explicitly or implicitly.
- Interfaces are not objects themselves; they provide a way to access an object. Therefore, clients do not access data directly; clients access data through an interface pointer (see page 1387). Windows 2000 adds an additional layer of indirection known as an interceptor through which it provides COM+ features such as just-in-time activation and object pooling.
- Interfaces are always inherited from the fundamental interface, IUnknown. (see page 1387)
- Interfaces can be redirected by COM through proxies to enable interface method calls to call between threads, processes, and networked machines, all without the client or server objects ever being aware of the redirection. For more information, see In-process (see page 1389).

See Also

The Fundamental COM Interface (see page 1387)

COM Interface Pointers (see page 1387)

3.2.2.1.4 The Fundamental COM Interface, IUnknown

All COM objects must support the fundamental interface, called *IUnknown, a typedefto the base interface type IInterface. IUnknown contains the following routines:*

QueryInterface	Provides pointers to other interfaces that the object supports.
AddRef and Release	Simple reference counting methods that keep track of the object's lifetime so that an object can delete itself when the client no longer needs its service.

Clients obtain pointers to other interfaces through the *IUnknown* method, *QueryInterface*. *QueryInterface* knows about every interface in the server object and can give a client a pointer to the requested interface. When receiving a pointer to an interface, the client is assured that it can call any method of the interface.

Objects track their own lifetime through the *IUnknown* methods, *AddRef* and *Release*, which are simple reference counting methods. As long as an object's reference count is nonzero, the object remains in memory. Once the reference count reaches zero, the interface implementation can safely dispose of the underlying object(s).

See Also

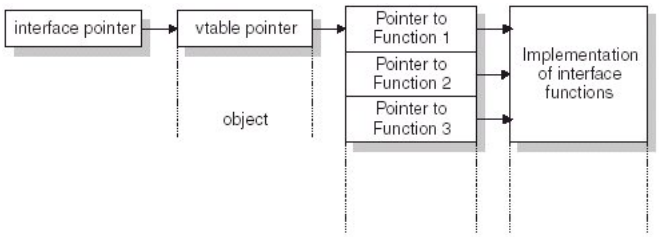
COM Interfaces (see page 1386)

COM Interface Pointers (see page 1387)

3.2.2.1.5 COM Interface Pointers

An interface pointer is a pointer to an object instance that points, in turn, to the implementation of each method in the interface. The implementation is accessed through an array of pointers to these methods, which is called a **vtable**. Vtables are similar to the mechanism used to support virtual functions in Delphi. Because of this similarity, the compiler can resolve calls to methods on the interface the same way it resolves calls to methods on Delphi classes.

The vtable is shared among all instances of an object class, so for each object instance, the object code allocates a second structure that contains its private data. The client's interface pointer, then, is a pointer *to the pointer* to the vtable, as shown in the following diagram.



In Windows 2000 and subsequent versions of Windows, when an object is running under COM+, an added level of indirection is provided between the interface pointer and the vtable pointer. The interface pointer available to the client points at an interceptor, which in turn points at the vtable. This allows COM+ to provide such services as just-in-time activation, whereby the server can be deactivated and reactivated dynamically in a way that is opaque to the client. To achieve this, COM+ guarantees that the interceptor behaves as if it were an ordinary vtable pointer.

See Also

The Fundamental COM Interface (see page 1387)

COM Interfaces (see page 1386)

3.2.2.1.6 COM Servers

A COM server is an application or a library that provides services to a client application or library. A COM server consists of one or more COM objects, where a COM object is a set of properties and methods.

Clients do not know *how* a COM object performs its service; the object's implementation remains encapsulated. An object makes its services available through its interfaces (see page 1386).

In addition, clients do not need to know *where* a COM object resides. COM provides transparent access regardless of the object's location. (see page 1389)

When a client requests a service from a COM object, the client passes a class identifier (CLSID) to COM. A CLSID is simply a GUID that identifies a COM object. COM uses this CLSID, which is registered in the system registry, to locate the appropriate server implementation. Once the server is located, COM brings the code into memory, and has the server instantiate an object instance for the client. This process is handled indirectly, through a special object called a class factory (based on interfaces) that creates instances of objects on demand.

As a minimum, a COM server must perform the following:

- Register entries in the system registry that associate the server module with the class identifier (CLSID).
- Implement a class factory object (see page 1388), which manufactures another object of a particular CLSID.
- Expose the class factory to COM.
- Provide an unloading mechanism through which a server that is not servicing clients can be removed from memory.

Note: Delphi wizards automate the creation of COM objects and servers (see page 1397).

See Also

COM Interfaces (see page 1386)

CoClasses and Class Factories (see page 1388)

In-process (see page 1389)

3.2.2.1.7 CoClasses and Class Factories

A COM object is an instance of a **CoClass**, which is a class that implements one or more COM interfaces. The COM object

provides the services as defined by its interfaces.

CoClasses are instantiated by a special type of object called a *class factory*. Whenever an object's services are requested by a client, a class factory creates an object instance for that particular client. Typically, if another client requests the object's services, the class factory creates another object instance to service the second client. (Clients can also bind to running COM objects that register themselves to support it.)

A CoClass must have a class factory and a class identifier (CLSID) so that it can be instantiated externally, that is, from another module. Using these unique identifiers for CoClasses means that they can be updated whenever new interfaces are implemented in their class. A new interface can modify or add methods without affecting older versions, which is a common problem when using DLLs.

Delphi wizards take care of assigning class identifiers and of implementing and instantiating class factories.

See Also

COM Servers (see page 1388)

In-process (see page 1389)

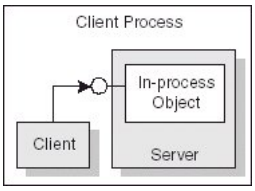
Aggregation (see page 1391)

3.2.2.1.8 In-process, Out-of-process, and Remote Servers

With COM, a client does not need to know where an object resides, it simply makes a call to an object's interface. COM performs the necessary steps to make the call. These steps differ depending on whether the object resides in the same process as the client, in a different process on the client machine, or in a different machine across the network. The different types of servers are known as:

In-process server	A library (DLL) running in the *same process space* as the client, for example, an ActiveX control embedded in a Web page viewed under Internet Explorer or Netscape. Here, the ActiveX control is downloaded to the client machine and invoked within the same process as the Web browser. The client communicates with the in-process server using direct calls to the COM interface.
Out-of-process server (or local server)	Another application (EXE) running in a *different process space* but on the *same machine* as the client. For example, an Excel spreadsheet embedded in a Word document are two separate applications running on the same machine. The local server uses COM to communicate with the client.
Remote server	A DLL or another application running on a *different machine* from that of the client. For example, a Delphi database application is connected to an application server on another machine in the network. The remote server uses distributed COM (DCOM) to access interfaces and communicate with the application server.

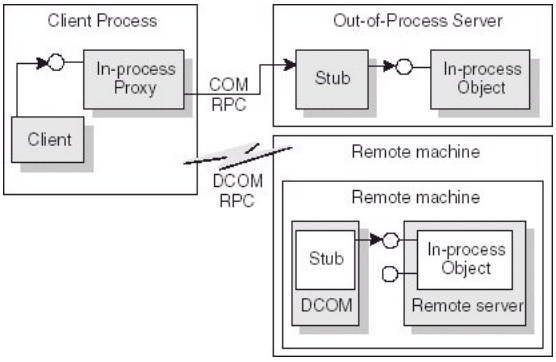
As shown in the following figure, for in-process servers, pointers to the object interfaces are in the same process space as the client, so COM makes direct calls into the object implementation.



Note: This is not always true under COM+. When a client makes a call to an object in a different context, COM+ intercepts the call so that it behaves like a call to an out-of-process server (see below), even if the server is in-process.

As shown in the following figure, when the process is either in a different process or in a different machine altogether, COM uses a proxy to initiate remote procedure calls. The **proxy** resides in the same process as the client, so from the client's

perspective, all interface calls look alike. The proxy intercepts the client's call and forwards it to where the real object is running. The mechanism that enables the client to access objects in a different process space, or even different machine, as if they were in their own process, is called marshaling (see page 1390).



The difference between out-of-process and remote servers is the type of interprocess communication used. The proxy uses COM to communicate with an out-of-process server, it uses distributed COM (DCOM) to communicate with a remote machine. DCOM transparently transfers a local object request to the remote object running on a different machine.

Note: For remote procedure calls, DCOM uses the RPC protocol provided by Open Group's Distributed Computing Environment (DCE). For distributed security, DCOM uses the NT LAN Manager (NTLM) security protocol. For directory services, DCOM uses the Domain Name System (DNS).

See Also

CoClasses and Class Factories (see page 1388)

COM Servers (see page 1388)

The Marshaling Mechanism (see page 1390)

Aggregation (see page 1391)

3.2.2.1.9 The Marshaling Mechanism

Marshaling is the mechanism that allows a client to make interface function calls to remote objects in another process or on a different machine. Marshaling

- Takes an interface pointer in the server's process and makes a proxy pointer available to code in the client process.
- Transfers the arguments of an interface call as passed from the client and places the arguments into the remote object's process space.

For any interface call, the client pushes arguments onto a stack and makes a function call through the interface pointer. If the call to the object is not in-process, the call gets passed to the proxy. The proxy packs the arguments into a marshaling packet and transmits the structure to the remote object. The object's stub unpacks the packet, pushes the arguments onto the stack, and calls the object's implementation. In essence, the object recreates the client's call in its own address space.

The type of marshaling that occurs depends on what interface the COM object implements. Objects can use a standard marshaling mechanism provided by the *IDispatch* interface. This is a generic marshaling mechanism that enables communication through a system-standard remote procedure call (RPC). For details on the *IDispatch* interface, see Automation Interfaces (see page 1438). Even if the object does not implement *IDispatch*, if it limits itself to automation-compatible types and has a registered type library, COM automatically provides marshaling support.

Applications that do not limit themselves to automation-compatible types or register a type library must provide their own marshaling. Marshaling is provided either through an implementation of the *IMarshal* interface, or by using a separately generated proxy/stub DLL. Delphi does not support the automatic generation of proxy/stub DLLs.

See Also

In-process (see page 1389)

Automation (see page 1393)

Aggregation (see page 1391)

3.2.2.1.10 Aggregation (COM)

Sometimes, a server object makes use of another COM object to perform some of its functions. For example, an inventory management object might make use of a separate invoicing object to handle customer invoices. If the inventory management object wants to present the invoice interface to clients, however, there is a problem: Although a client that has the inventory interface can call *QueryInterface* to obtain the invoice interface, when the invoice object was created it did not know about the inventory management object and can't return an inventory interface in response to a call to *QueryInterface*. A client that has the invoice interface can't get back to the inventory interface.

To avoid this problem, some COM objects support **aggregation**. When the inventory management object creates an instance of the invoice object, it passes it a copy of its own *IUnknown* interface. The invoice object can then use that *IUnknown* interface to handle any *QueryInterface* calls that request an interface, such as the inventory interface, that it does not support. When this happens, the two objects together are called an aggregate. The invoice object is called the inner, or contained object of the aggregate, and the inventory object is called the outer object.

Note: In order to act as the outer object of an aggregate, a COM object must create the inner object using the Windows API *CoCreateInstance* or *CoCreateInstanceEx*, passing its *IUnknown* pointer as a parameter that the inner object can use for *QueryInterface* calls.

In order to create an object that can act as the inner object of an aggregate, it must descend from TContainedObject. When the object is created, the *IUnknown* interface of the outer object is passed to the constructor so that it can be used by the *QueryInterface* method on calls that the inner object can't handle.

See Also

In-process (see page 1389)

Automation (see page 1393)

The Marshaling Mechanism (see page 1390)

3.2.2.1.11 COM Clients

Clients can always query the interfaces of a COM object to determine what it is capable of providing. All COM objects allow clients to request known interfaces. In addition, if the server supports the *IDispatch* interface, clients can query the server for information about what methods the interface supports. Server objects have no expectations about the client using its objects. Similarly, clients don't need to know how (or even where) an object provides the services; they simply rely on server objects to provide the services they advertise through their interfaces.

There are two types of COM clients, controllers and containers. Controllers launch the server and interact with it through its interface. They request services from the COM object or drive it as a separate process. Containers host visual controls or objects that appear in the container's user interface. They use predefined interfaces to negotiate display issues with server objects. It is impossible to have a container relationship over DCOM; for example, visual controls that appear in the container's user interface must be located locally. This is because the controls are expected to paint themselves, which requires that they have access to local GDI resources.

Delphi makes it easier for you to develop COM clients by letting you import a type library or ActiveX control into a component wrapper so that server objects look like other VCL components. For details on this process, see Creating COM clients (see

page 1419)

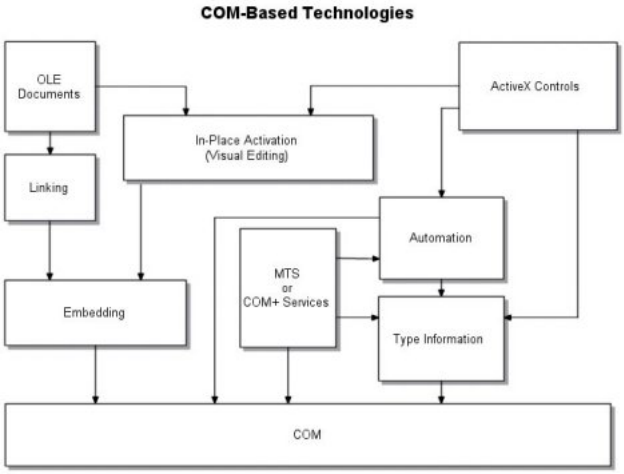
3.2.2.1.12 COM Extensions

COM was originally designed to provide core communication functionality and to enable the broadening of this functionality through extensions. COM itself has extended its core functionality by defining specialized sets of interfaces for specific purposes.

The following lists some of the services COM extensions currently provide.

Automation servers (see page 1393)	Automation refers to the ability of an application to control the objects in another application programmatically. Automation servers are the objects that can be controlled by other executables at runtime.
ActiveX controls (see page 1394)	ActiveX controls are specialized in-process servers, typically intended for embedding in a client application. The controls offer both design and runtime behaviors as well as events.
Active Server Pages (see page 1393)	Active Server Pages are scripts that generate HTML pages. The scripting language includes constructs for creating and running Automation objects. That is, the Active Server Page acts as an Automation controller.
Active Documents (see page 1394)	Objects that support linking and embedding, drag-and-drop, visual editing, and in-place activation. Word documents and Excel spreadsheets are examples of Active Documents.
COM+ Event and event subscription objects	Objects that support the loosely coupled COM+ Events model. Unlike the tightly coupled model used by ActiveX controls, the COM+ Events model allows you to develop event publishers independently of event subscribers.
Type libraries (see page 1395)	A collection of static data structures, often saved as a resource, that provides detailed type information about an object and its interfaces. Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available.

The following diagram illustrates the relationship of the COM extensions and how they are built upon COM:



COM objects can be visual or non-visual. Some must run in the same process space as their clients; others can run in different processes or remote machines, as long as the objects provide marshaling support. The following table summarizes the types of COM objects that you can create, whether they are visual, process spaces they can run in, how they provide marshaling (see page 1390), and whether they require a type library.

COM object requirements

Object	Visual Object?	Process space	Communication	Type library
Active Document	Usually	In-process, or out-of-process	OLE Verbs	No
Automation Server	Occasionally	In-process, out-of-process, or remote	Automatically marshaled using the *IDispatch* interface (for out-of process and remote servers)	Required for automatic marshaling
ActiveX Control	Usually	In-process	Automatically marshaled using the *IDispatch* interface	Required
COM+	Occasionally	In-process for MTS,any for COM+	Automatically marshaled via a type library	Required
In-process custom interface object	Optionally	In-process	No marshaling required for in-process servers	Recommended
Other custom interface object	Optionally	In-process,out-of-process, or remote	Automatically marshaled via a type library; otherwise, manually marshaled using custom interfaces	Recommended

3.2.2.1.13 Automation Servers

Automation refers to the ability of an application to control the objects in another application programmatically, like a macro that can manipulate more than one application at the same time. The server object being manipulated is called the Automation object, and the client of the Automation object is referred to as an Automation controller.

Automation can be used on in-process, local, and remote servers.

Automation is characterized by two key points:

- The Automation object defines a set of properties and commands, and describes their capabilities through type descriptions. In order to do this, it must have a way to provide information about its interfaces, the interface methods, and those methods' arguments. Typically, this information is available in a type library (see page 1395). The Automation server can also generate type information dynamically when queried via its *IDispatch* interface (see following).
- Automation objects make their methods accessible so that other applications can use them. For this, they implement the *IDispatch* interface. Through this interface an object can expose all of its methods and properties. Through the primary method of this interface, the object's methods can be invoked, once having been identified through type information.

Developers often use Automation to create and use non-visual OLE objects that run in any process space because the Automation *IDispatch* interface automates the marshaling process. Automation does, however, restrict the types that you can use.

For a list of types that are valid for type libraries in general, and Automation interfaces in particular, see Valid types (see page 1454).

See Also

Creating Simple COM Servers: Overview (see page 1430)

Working with Type Libraries: Overview (see page 1445)

Creating Active Server Pages (see page 1400)

3.2.2.1.14 Active Server Pages

The Active Server Page (ASP) technology lets you write simple scripts, called Active Server Pages, that can be launched by clients via a Web server. Unlike ActiveX controls, which run on the client, Active Server Pages run on the server, and return a

resulting HTML page to clients.

Active Server Pages are written in Jscript or VB script. The script runs every time the server loads the Web page. That script can then launch an embedded Automation server (or Enterprise Java Bean). For example, you can write an Automation server, such as one to create a bitmap or connect to a database, and this server accesses data that gets updated every time a client loads the Web page.

Active Server Pages rely on the Microsoft Internet Information Server (IIS) environment to serve your Web pages.

Delphi wizards let you Create Active Server Pages (see page 1400), which is an Automation object specifically designed to work with an Active Server Page.

See Also

Creating Active Server Pages (see page 1400)

Working with Type Libraries: Overview (see page 1445)

3.2.2.1.15 ActiveX Controls

ActiveX is a technology that allows COM components, especially controls, to be more compact and efficient. This is especially necessary for controls that are intended for Intranet applications that need to be downloaded by a client before they are used.

ActiveX controls are visual controls that run only as in-process servers, and can be plugged into an ActiveX control container application. They are not complete applications in themselves, but can be thought of as prefabricated OLE controls that are reusable in various applications. ActiveX controls have a visible user interface, and rely on predefined interfaces to negotiate I/O and display issues with their host containers.

ActiveX controls make use of Automation to expose their properties, methods, and events. Features of ActiveX controls include the ability to fire events, bind to data sources, and support licensing.

One use of ActiveX controls is on a Web site as interactive objects in a Web page. As such, ActiveX is a standard that targets interactive content for the World Wide Web, including the use of ActiveX Documents used for viewing non-HTML documents through a Web browser. For more information about ActiveX technology, see the Microsoft ActiveX Web site.

See Also

Using ActiveX Controls (see page 1406)

3.2.2.1.16 Active Documents

Active Documents (previously referred to as OLE documents) are a set of COM services that support linking and embedding, drag-and-drop, and visual editing. Active Documents can seamlessly incorporate data or objects of different formats, such as sound clips, spreadsheets, text, and bitmaps.

Unlike ActiveX controls, Active Documents are not limited to in-process servers; they can be used in cross-process applications.

Unlike Automation objects, which are almost never visual, Active Document objects can be visually active in another application. Thus, Active Document objects are associated with two types of data: presentation data, used for visually displaying the object on a display or output device, and native data, used to edit an object.

Active Document objects can be document containers or document servers. While Delphi does not provide an automatic wizard for creating Active Documents, you can use the VCL class, TOleContainer, to support linking and embedding of existing Active Documents.

You can also use *TOleContainer* as a basis for an Active Document container. To create objects for Active Document servers, use the COM object wizard and add the appropriate interfaces, depending on the services the object needs to support. For more information about creating and using Active Document servers, see the Microsoft ActiveX Web site.

Note: While the specification for Active Documents has built-in support for marshaling in cross-process applications, Active Documents do not run on remote servers because they use types that are specific to a system on a given machine such as window handles, menu handles, and so on.

3.2.2.1.17 Type Libraries

Type libraries provide a way to get more type information about an object than can be determined from an object's interface. The type information contained in type libraries provides needed information about objects and their interfaces, such as what interfaces exist on what objects (given the CLSID), what member functions exist on each interface, and what arguments those functions require.

You can obtain type information either by querying a running instance of an object or by loading and reading type libraries. With this information, you can implement a client which uses a desired object, knowing specifically what member functions you need, and what to pass those member functions.

Clients of Automation servers, ActiveX controls, and transactional objects expect type information to be available. All of Delphi's wizards generate a type library automatically, although the COM object wizard makes this optional. You can view or edit this type information by using the Type Library Editor (see page 1445). The Type Library Editor displays the type information by way of an intermediate text-based format (the RIDL file (see page 1468)) and generates the corresponding .tlb file when you build or save your project. You can also use the GenTLB.exe (see page 1468) utility to create a .tlb file from a RIDL file.

The content of type libraries

Type libraries contain *type information*, which indicates which interfaces exist in which COM objects, and the types and numbers of arguments to the interface methods. These descriptions include the unique identifiers for the CoClasses (CLSIDs) and the interfaces (IIDs), so that they can be properly accessed, as well as the dispatch identifiers (dispIDs) for Automation interface methods and properties.

Type libraries can also contain the following information:

- Descriptions of custom type information associated with custom interfaces
- Routines that are exported by the Automation or ActiveX server, but that are not interface methods
- Information about enumeration, record (structures), unions, alias, and module data types
- References to type descriptions from other type libraries

Creating type libraries

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then running that script through a compiler. However, Delphi automatically generates a type library when you create a COM object (including ActiveX controls, Automation objects, remote data modules, and so on) using any of the wizards on either the ActiveX or Multitier page of the new items dialog. You can also create a type library by choosing **File** ▸ **New** ▸ **Other**, selecting the ActiveX folder under Delphi Projects or C++Builder Projects, and in the right pane choosing Type Library.

You can view and edit the type library using Delphi's **Type Library Editor**. Delphi automatically updates the corresponding .tlb file (binary type library file) when the type library is saved. For any changes to Interfaces and CoClasses that were created using a wizard, the Type Library editor also updates your implementation files.

When to use type libraries

It is important to create a type library for each set of objects that is exposed to external users, for example,

- ActiveX controls require a type library, which must be included as a resource in the DLL that contains the ActiveX controls.
- Exposed objects that support vtable binding of custom interfaces must be described in a type library because vtable references are bound at compile time. Clients import information about the interfaces from the type library and use that information to compile. For more information about vtable and compile time binding, see Automation interfaces (see page

1438).

- Applications that implement Automation servers should provide a type library so that clients can early bind to it.
- Objects instantiated from classes that support the *IProvideClassInfo* interface, such as all descendants of the VCL *TTypedComObject* class, must have a type library.
- Type libraries are not required, but are useful for identifying the objects used with OLE drag-and-drop.

When defining interfaces for internal use only (within an application) you do not need to create a type library. .

Accessing type libraries

The binary type library is normally a part of a resource file (.res) or a stand-alone file with a .tlb file-name extension. When included in a resource file, the type library can be bound into a server (.dll, .ocx, or .exe).

Once a type library has been created, object browsers, compilers, and similar tools can access type libraries through special type interfaces:

Special Type Interfaces

Interface	Description
ITypeLib	Provides methods for accessing a library of type descriptions.
ITypeLib2	Augments *ITypeLib* to include support for documentation strings, custom data, and statistics about the type library.
ITypeInfo	Provides descriptions of individual objects contained in a type library. For example, a browser uses this interface to extract information about objects from the type library.
ITypeInfo2	Augments *ITypeInfo* to access additional type library information, including methods for accessing custom data elements.
ITypeComp	Provides a fast way to access information that compilers need when binding to an interface.

Delphi can import and use type libraries from other applications by choosing Component|Import Component. Most of the VCL classes used for COM applications support the essential interfaces that are used to store and retrieve type information from type libraries and from running instances of an object. The VCL class *TTypedComObject* supports interfaces that provide type information, and is used as a foundation for the ActiveX object framework.

Benefits of using type libraries

Even if your application does not require a type library, you can consider the following benefits of using one:

- Type checking can be performed at compile time.
- You can use early binding with Automation, and controllers that do not support vtables or dual interfaces can encode dispIDs at compile time, improving runtime performance.
- Type browsers can scan the library, so clients can see the characteristics of your objects.
- The *RegisterTypeLib* function can be used to register your exposed objects in the registration database.
- The *UnRegisterTypeLib* function can be used to completely uninstall an application's type library from the system registry.
- Local server access is improved because Automation uses information from the type library to package the parameters that are passed to an object in another process.
- GenTLB.exe is a utility provided by CodeGear that generates a .tlb file from a RIDL file (an intermediate text-based file used by the Type Library Editor).

Using type library tools

The tools for working with type libraries are listed below.

- The TLIBIMP (Type Library Import) tool, which takes existing type libraries and creates Delphi Interface files (_TLB.pas files), is incorporated into the Type Library editor. TLIBIMP provides additional configuration options not available inside the Type Library editor.

- TRegSvr is a tool for registering and unregistering servers and type libraries, which comes with Delphi. The source to TRegSvr is available as an example in the Demos directory.
- The Microsoft IDL compiler (MIDL) compiles IDL scripts to create a type library.
- RegSvr32.exe is a standard Windows utility for registering and unregistering servers and type libraries.
- OLEView is a type library browser tool, found on Microsoft's Web site.

See Also

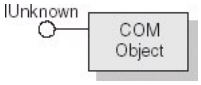
Working with Type Libraries (see page 1445)

3.2.2.1.18 Implementing COM Objects with Wizards

Delphi makes it easier to write COM server applications by providing wizards that handle many of the details involved. Delphi provides separate wizards to create the following:

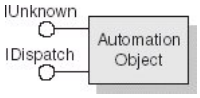
- A simple COM object
- An Automation object
- A COM+ Event Object
- A Type library
- An ActiveX library

The wizards handle many of the tasks involved in creating each type of COM object. They provide the required COM interfaces for each type of object. With a simple COM object, the wizard implements the one required COM interface, *IUnknown*, which provides an interface pointer to the object.

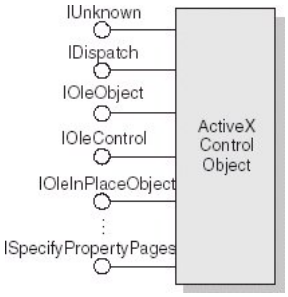


The COM object wizard also provides an implementation for *IDispatch* if you specify that you are creating an object that supports an *IDispatch* descendant.

For Automation and Active Server objects, the wizard implements *IUnknown* and *IDispatch*, which provides automatic marshaling.



For ActiveX control objects and ActiveX forms, the wizard implements all the required ActiveX control interfaces, from *IUnknown*, *IDispatch*, *IOleObject*, *IOleControl*, and so on. For a complete list of interfaces, see the reference page for TActiveXControl object.



The following table lists the various wizards and the interfaces they implement:

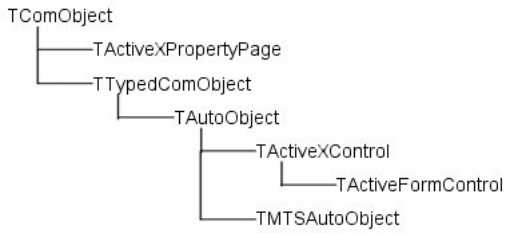
Delphi wizards for implementing COM, Automation, and ActiveX objects

Wizard	Implemented interfaces	What the wizard does
COM server (see page 1430)	*IUnknown* (and *IDispatch* if you select a default interface that descends from *IDispatch*)	Exports routines that handle server registration, class registration, loading and unloading the server, and object instantiation. Creates and manages class factories for objects implemented on the server. Provides registry entries for the object that specify the selected threading model. Declares the methods that implement a selected interface, providing skeletal implementations for you to complete. Provides a type library, if requested. Allows you to select an arbitrary interface that is registered in the type library and implement it. If you do this, you must use a type library.
Automation server (see page 1432)	*IUnknown*, *IDispatch*	Performs the tasks of a COM server wizard (described above), plus: Implements the interface that you specify, either dual or dispatch. Provides server-side support for generating events, if requested. Provides a type library automatically.
COM+ Event object	None, by default	Creates a COM+ event object that you can define using the Type Library editor. Unlike the other object wizards, the COM+ Event object wizard does not create an implementation unit because event objects have no implementation (it is provided by event subscriber objects).
Type Library (see page 1445)	None, by default	Creates a new type library and associates it with the active project.
ActiveX library	None, by default	Creates a new ActiveX or Com server DLL and exposes the necessary export functions.

You can add additional COM objects or reimplement an existing implementation. To add a new object, it is easiest to use the wizard a second time. This is because the wizard sets up an association between the type library and an implementation class, so that changes you make in the type library editor are automatically applied to your implementation object.

3.2.2.1.19 Code Generated by Wizards

Delphi's wizards generate classes that are derived from the Delphi ActiveX framework (DAX). Despite its name, the Delphi ActiveX framework supports all types of COM objects, not just ActiveX controls. The classes in this framework provide the underlying implementation of the standard COM interfaces for the objects you create using a wizard. The following figure illustrates the objects in the Delphi ActiveX framework:



Each wizard generates an implementation unit that implements your COM server object. The COM server object (the implementation object) descends from one of the classes in DAX:

DAX Base classes for generated implementation classes

Wizard	Base class from DAX	Inherited support
COM server (see page 1430)	TTypedComObject	Support for *IUnknown* and *ISupportErrorInfo* interfaces. Support for aggregation, OLE exception handling, and safecall calling convention on dual interfaces. Support for reading type library information.
Automation server (see page 1432) or Creating Active Server Pages (see page 1400)	TAutoObject	Everything provided by *TTypedComObject*, plus: Support for the IDispatch interface. Auto-marshaling support.

Corresponding to the classes in DAX is a hierarchy of class factory objects that handle the creation of these COM objects. The wizard adds code to the initialization section of your implementation unit that instantiates the appropriate class factory for your implementation class.

The wizards also generate a type library and its associated unit, which has a name of the form `Project1_TLB`. The `Project1_TLB` unit includes the definitions your application needs to use the type definitions and interfaces defined in the type library. For more information on the contents of this file, see Code generated when you import type library information (see page 1421).

You can modify the interface generated by the wizard using the type library editor. When you do this, the implementation class is automatically updated to reflect those changes. You need only fill in the bodies of the generated methods to complete the implementation.

3.2.2.1.20 COM+ Event And Event Subscriber Objects

The COM+ Events system introduces a middle layer of software that decouples applications that generate events (called publishers) from applications that respond to events (called subscribers). Instead of being tightly bound to each other, publishers and subscribers can be developed, deployed and activated independently of each other.

In the COM+ Events model, an event interface is first created using the COM+ Event Object wizard. The event interface has no implementation; it simply defines the event methods that publishers will generate, and that subscribers will respond to. The COM+ event object is then installed into a COM+ Application, in the COM+ Catalog. This can be done programatically using the TComAdminCatalog object, or by a system administrator using the Component Services tool.

Event subscribers are responsible for providing an implementation for the event interface. You can create event subscriber components using the COM+ Event Subscription wizard. Using the wizard, you can select the event object you want to implement, and the IDE will stub out each method of the interface. You can also select a type library if the event object has not yet been installed in the COM+ Catalog.

Finally, once the subscriber component has been implemented, it too must be installed in the COM+ Catalog. Again, this can be done with a TComAdminCatalog object, or by using the Component Services administrative tool.

When a publisher wishes to generate an event, it simply creates an instance of the event object (not the subscriber component), and calls the appropriate methods on the event interface. COM+ then steps in and notifies all applications that have subscribed to that event object, calling them synchronously, one at a time. This way, publishers need not know anything about those applications that are subscribing to the event. Subscribers don't need anything more than an implementation of the event interface, and to select those publishers they wish to subscribe to. COM+ handles the rest.

For more information regarding the COM+ Events system, see Generating events under COM+.

3.2.2.2 Creating an Active Server Page

Topics

Name	Description
Creating Active Server Pages: Overview (see page 1400)	If you are using the Microsoft Internet Information Server (IIS) environment to serve your Web pages, you can use Active Server Pages (ASP) to create dynamic Web-based client/server applications. Active Server Pages let you write a script that gets called every time the server loads the Web page. This script can, in turn, call on Automation objects (see page 1430) to obtain information that it includes in a generated HTML page. For example, you can write a Delphi Automation server, such as one to create a bitmap or connect to a database, and use this control to access data that gets updated every... more (see page 1400)
Creating an Active Server Object (see page 1401)	An Active Server Object is an Automation object that has access to information about the entire ASP application and the HTTP messages it uses to communicate with browsers. It descends from TASPObject or TASPMTSObject (which is in turn a descendant of TAutoObject), and supports Automation protocols, exposing itself for other applications (or the script in the Active Server page) to use. You create an Active Server Object using the Active Server Object wizard. Your Active Server Object project can be either an executable (exe) or library (dll), depending on your needs. However, you should be aware of the drawbacks of (see page 1405)... more (see page 1401)
Using the ASP Intrinsics (see page 1402)	The ASP intrinsics are a set of COM objects supplied by ASP to the objects running in an Active Server Page. They let your Active Server Object access information that reflects the messages passing between your application and the Web browser, as well as a place to store information that is shared among Active Server Objects that belong to the same ASP application. To make these objects easy to access, the base class for your Active Server Object surfaces them as properties. For a complete understanding of these objects, see the Microsoft documentation. However, the following topics provide a brief... more (see page 1402)
Creating ASPs for In-process or Out-of-process Servers (see page 1405)	You can use **Server.CreateObject** in an ASP page to launch either an in-process or out-of-process (see page 1389) server, depending on your requirements. However, launching in-process servers is more common. Unlike most in-process servers, an Active Server Object in an in-process server does not run in the client's process space. Instead, it runs in the IIS process space. This means that the client does not need to download your application (as, for example, it does when you use ActiveX objects). In-process component DLLs are faster and more secure than out-of-process servers, so they are better suited for server-side use. Because out-of-process servers are... more (see page 1405)
Registering an Active Server Object (see page 1405)	You can register the Active Server Page as an in-process or an out-of-process server (see page 1405). However, in-process servers are more common. **Note:** When you want to remove the Active Server Page object from your system, you should first unregister it, removing its entries from the Windows registry.
Testing and Debugging the Active Server Page Application (see page 1406)	Debugging any in-process server such as an Active Server Object is much like debugging a DLL. You choose a host application that loads the DLL, and debug as usual.

3.2.2.2.1 Creating Active Server Pages: Overview

If you are using the Microsoft Internet Information Server (IIS) environment to serve your Web pages, you can use Active Server Pages (ASP) to create dynamic Web-based client/server applications. Active Server Pages let you write a script that gets called every time the server loads the Web page. This script can, in turn, call on Automation objects (see page 1430) to obtain information that it includes in a generated HTML page. For example, you can write a Delphi Automation server, such as one to create a bitmap or connect to a database, and use this control to access data that gets updated every time the server loads the Web page.

On the client side, the ASP acts like a standard HTML document and can be viewed by users on any platform using any Web Browser.

ASP applications are analogous to applications you write using Delphi's Web broker technology. For more information about the Web broker technology, see Creating Internet server applications (see page 2251). ASP differs, however, in the way it separates the UI design from the implementation of business rules or complex application logic.

- The UI design is managed by the Active Server Page. This is essentially an HTML document, but it can include embedded script that calls on Active Server objects to supply it with content that reflects your business rules or application logic.
- The application logic is encapsulated by Active Server objects that expose simple methods to the Active Server Page, supplying it with the content it needs.

Note: Although ASP provides the advantage of separating UI design from application logic, its performance is limited in scale. For Web sites that respond to extremely large numbers of clients, an approach based on the Web broker technology is recommended instead.

The script in your Active Server Pages and the Automation objects you embed in an active server page can make use of the ASP intrinsics (see page 1402) (built-in objects that provide information about the current application, HTTP messages from the browser, and so on).

The following topics show how to create an Active Server Object using the Delphi Active Server Object wizard. This special Automation control can then be called by an Active Server Page and supply it with content.

Here are the steps for creating an Active Server Object:

- Create an Active Server Object (see page 1401) for the application.
- Define the Active Server Object's interface. (see page 1435)
- Register (see page 1405) the Active Server Object.
- Test and debug (see page 1406) the application.

See Also

Overview of COM Technologies (see page 1385)

Creating Simple COM Servers: Overview (see page 1430)

3.2.2.2.2 Creating an Active Server Object

An Active Server Object is an Automation object that has access to information about the entire ASP application and the HTTP messages it uses to communicate with browsers. It descends from TASPObject or TASPMTSObject (which is in turn a descendant of TAutoObject), and supports Automation protocols, exposing itself for other applications (or the script in the Active Server page) to use. You create an Active Server Object using the Active Server Object wizard.

Your Active Server Object project can be either an executable (exe) or library (dll), depending on your needs. However, you should be aware of the drawbacks of using an out-of-process server (see page 1405).

To display the Active Server Object wizard:

1. Choose **File** ▸ **New** ▸ **Other**.
2. Select the folder labeled ActiveX under Delphi Projects.
3. Double-click the Active Server Object icon. In the wizard, give your new Active Server Object a name, and specify the instancing (see page 1433) and threading (see page 1433) models you want to support. These details influence the way your object can be called. You must write the implementation so that it adheres to the model (for example, avoiding thread conflicts). The thing that makes an Active Server Object unique is its ability to access information about the ASP application and the HTTP messages that pass between the Active Server page and client Web browsers. This information is accessed using the ASP intrinsics (see page 1402). In the wizard, you can specify how your object accesses these by setting the Active Server Type:

- If you are working with IIS 3 or IIS 4, you use Page Level Event Methods. Under this model, your object implements the methods, OnStartPage and OnEndPage, which are called when the Active Server page loads and unloads. When your object is loaded, it automatically obtains an *IScripti*ngContext interface, which it uses to access the ASP intrinsics. These interfaces are, in turn, surfaced as properties inherited from the base class (*TASPObject*).

- If you are working with IIS5 or later, you use the Object Context type. Under this model, your object fetches an IObjectContext interface, which it uses to access the ASP intrinsics. Again, these interfaces are surfaced as properties in the inherited base class (TASPMTSObject). One advantage of this latter approach is that your object has access to all of the other services available through *IObjectContext*. To access the *IObjectContext* interface, simply call *GetObjectContext* (defined in the mtx unit) as follows: `ObjectContext := GetObjectContext;` For more information about the services available through *IObjectContext*, see Creating MTS or COM+ objects

You can tell the wizard to generate a simple ASP page to host your new Active Server Object. The generated page provides a minimal script (written in VBScript) that creates your Active Server Object based on its ProgID, and indicates where you can call its methods. This script calls **Server.CreateObject** to launch your Active Server Object.

Note: Although the generated test script uses VBScript, Active Server Pages also can be written using Jscript.

When you exit the wizard, a new unit is added to the current project that contains the definition for the Active Server Object. In addition, the wizard adds a type library project and opens the Type Library editor. Now you can expose the properties and methods of the interface through the type library as described in Defining a COM object's interface (see page 1435) As you write the implementation of your object's properties and methods, you can take advantage of the ASP intrinsics (see page 1402) to obtain information about the ASP application and the HTTP messages it uses to communicate with browsers.

The Active Server Object, like any other Automation object, implements a dual interface (see page 1438), which supports both early (compile-time) binding through the VTable and late (runtime) binding through the IDispatch interface.

See Also

Implementing COM objects with Wizards (see page 1397)

Using the Automation Object Wizard (see page 1432)

Code Generated by Wizards (see page 1398)

Creating Simple COM Servers: Overview (see page 1430)

3.2.2.2.3 Using the ASP Intrinsics

The ASP intrinsics are a set of COM objects supplied by ASP to the objects running in an Active Server Page. They let your Active Server Object access information that reflects the messages passing between your application and the Web browser, as well as a place to store information that is shared among Active Server Objects that belong to the same ASP application.

To make these objects easy to access, the base class for your Active Server Object surfaces them as properties. For a complete understanding of these objects, see the Microsoft documentation. However, the following topics provide a brief overview.

Application

The Application object is accessed through an *IApplicat*ionObject interface. It represents the entire ASP application, which is defined as the set of all .asp files in a virtual directory and its subdirectories. The Application object can be shared by multiple clients, so it includes locking support that you should use to prevent thread conflicts.

IApplicationObject includes the following:

IApplicationObject interface members

Property, Method, or Event	Meaning
Contents property	Lists all the objects that were added to the application using script commands. This interface has two methods, *Remove* and *RemoveAll*, that you can use to delete one or all objects from the list.
StaticObjects property	Lists all the objects that were added to the application with the <OBJECT> tag.
Lock method	Prevents other clients from locking the Application object until you call Unlock. All clients should call Lock before accessing shared memory (such as the properties).

Unlock method	Releases the lock that was set using the Lock method.
Application_OnEnd event	Occurs when the application quits, after the Session_OnEnd event. The only intrinsics available are Application and Server. The event handler must be written in VBScript or JScript.
Application_OnStart event	Occurs before the new session is created (before Session_OnStart). The only intrinsics available are Application and Server. The event handler must be written in VBScript or JScript.

Request

The Request object is accessed through an *IReque*st interface. It provides information about the HTTP request message that caused the Active Server Page to be opened.

IRequest includes the following:

IRequest interface members

Property, Method, or Event	Meaning
ClientCertificate property	Indicates the values of all fields in the client certificate that is sent with the HTTP message.
Cookies property	Indicates the values of all Cookie headers on the HTTP message.
Form property	Indicates the values of form elements in the HTTP body. These can be accessed by name.
QueryString property	Indicates the values of all variables in the query string from the HTTP header.
ServerVariables property	Indicates the values of various environment variables. These variables represent most of the common HTTP header variables.
TotalBytes property	Indicates the number of bytes in the request body. This is an upper limit on the number of bytes returned by the BinaryRead method.
BinaryRead method	Retrieves the content of a Post message. Call the method, specifying the maximum number of bytes to read. The resulting content is returns as a Variant array of bytes. After calling BinaryRead, you can't use the Form property.

Response

The Request object is accessed through an *IRe*sponse interface. It lets you specify information about the HTTP response message that is returned to the client browser.

IResponse includes the following:

IResponse interface members

Property, Method, or Event	Meaning
Cookies property	Determines the values of all Cookie headers on the HTTP message.
Buffer property	Indicates whether page output is buffered When page output is buffered, the server does not send a response to the client until all of the server scripts on the current page are processed.
CacheControl property	Determines whether proxy servers can cache the output in the response.
Charset property	Adds the name of the character set to the content type header.
ContentType property	Specifies the HTTP content type of the response message's body.
Expires property	Specifies how long the response can be cached by a browser before it expires.
ExpiresAbsolute property	Specifies the date and time when the response expires.
IsClientConnected property	Indicates whether the client has disconnected from the server.

Pics property	Set the value for the pics-label field of the response header.
Status property	Indicates the status of the response. This is the value of an HTTP status header.
AddHeader method	Adds an HTTP header with a specified name and value.
AppendToLog method	Adds a string to the end of the Web server log entry for this request.
BinaryWrite method	Writes raw (uninterpreted) information to the body of the response message.
Clear method	Erases any buffered HTML output.
End method	Stops processing the .asp file and returns the current result.
Flush method	Sends any buffered output immediately.
Redirect method	Sends a redirect response message, redirecting the client browser to a different URL.
Write method	Writes a variable to the current HTTP output as a string.

Session

The Session object is accessed through the *ISessionObject* interface. It allows you to store variables that persist for the duration of a client's interaction with the ASP application. That is, these variables are not freed when the client moves from page to page within the ASP application, but only when the client exits the application altogether.

ISessionObject includes the following:

ISessionObject interface members

Property, Method, or Event	Meaning
Contents property	Lists all the objects that were added to the session using the <OBJECT> tag. You can access any variable in the list by name, or call the Contents object's *Remove* or *RemoveAll* method to delete values.
StaticObjects property	Lists all the objects that were added to the session with the <OBJECT> tag.
CodePage property	Specifies the code page to use for symbol mapping. Different locales may use different code pages.
LCID property	Specifies the locale identifier to use for interpreting string content.
SessionID property	Indicates the session identifier for the current client.
TimeOut property	Specifies the time, in minutes, that the session persists without a request (or refresh) from the client until the application terminates.
Abandon method	Destroys the session and releases its resources.
Session_OnEnd event	Occurs when the session is abandoned or times out. The only intrinsics available are Application, Server, and Session. The event handler must be written in VBScript or JScript.
Session_OnStart event	Occurs when the server creates a new session is created (after Application_OnStart but before running the script on the Active Server Page). All intrinsics are available. The event handler must be written in VBScript or JScript.

Server

The Server object is accessed through an *IServer* interface. It provides various utilities for writing your ASP application.

IServer includes the following:

IServer interface members

Property, Method, or Event	Meaning
ScriptTimeOut property	Same as the TimeOut property on the Session object.
CreateObject method	Instantiates a specified Active Server Object.
Execute method	Executes the script in a specified .asp file.
GetLastError method	Returns an ASPError object that describes the error condition.
HTMLEncode method	Encodes a string for use in an HTML header, replacing reserved characters by the appropriate symbolic constants.
MapPath method	Maps a specified virtual path (an absolute path on the current server or a path relative to the current page) into a physical path.
Transfer method	Sends all of the current state information to another Active Server Page for processing.
URLEncode method	Applies URL encoding rules, including escape characters, to a specified string

See Also

Creating ASPs for In-process or Out-of-process Servers (see page 1405)

Defining a COM Object's Interface (see page 1435)

3.2.2.2.4 Creating ASPs for In-process or Out-of-process Servers

You can use **Server.CreateObject** in an ASP page to launch either an in-process or out-of-process (see page 1389) server, depending on your requirements. However, launching in-process servers is more common.

Unlike most in-process servers, an Active Server Object in an in-process server does not run in the client's process space. Instead, it runs in the IIS process space. This means that the client does not need to download your application (as, for example, it does when you use ActiveX objects). In-process component DLLs are faster and more secure than out-of-process servers, so they are better suited for server-side use.

Because out-of-process servers are less secure, it is common for IIS to be configured to *not* allow out-of-process executables. In this case, creating an out-of-process server for your Active Server Object would result in an error similar to the following:

```
Server object error 'ASP 0196'
Cannot launch out of process component
/path/outofprocess_exe.asp, line 11
```

Also, out-of-process components often create individual server processes for each object instance, so they are slower than CGI applications. They do not scale as well as component DLLs.

If performance and scalability are priorities for your site, in-process servers are highly recommended. However, Intranet sites that receive moderate to low traffic may use an out-of-process component without adversely affecting the site's overall performance.

See Also

Implementing COM objects with Wizards (see page 1397)

Creating Simple COM Servers: Overview (see page 1430)

3.2.2.2.5 Registering an Active Server Object

You can register the Active Server Page as an in-process or an out-of-process server (see page 1405). However, in-process servers are more common.

Note: When you want to remove the Active Server Page object from your system, you should first unregister it, removing its entries from the Windows registry.

Registering an in-process server

To register an in-process server (DLL or OCX), choose **Run ▶ Register ActiveX Server**.

To unregister an in-process server, choose **Run ▶ Unregister ActiveX Server**.

Registering an out-of-process server

To register an out-of-process server, run the server with the /regserver command-line option. You can also register the server by running it.

To unregister an out-of-process server, run the server with the /unregserver command-line option.

3.2.2.2.6 Testing and Debugging the Active Server Page Application

Debugging any in-process server such as an Active Server Object is much like debugging a DLL. You choose a host application that loads the DLL, and debug as usual.

To test and debug an Active Server Object,

1. Turn on debugging information using the Compiler tab on the **Project ▶ Options** dialog box, if necessary. Also, turn on Integrated Debugging in the **Tools ▶ Options ▶ Debugger Options** dialog.
2. Choose **Run ▶ Parameters**, type the name of your Web Server in the Host Application box, and choose OK.
3. Choose **Run ▶ Run**.
4. Set breakpoints in the Active Server Object implementation.
5. Use the Web browser to interact with the Active Server Page.

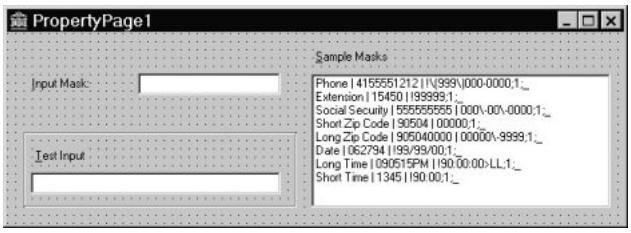
The debugger pauses when the breakpoints are reached.

3.2.2.3 Using ActiveX controls

Topics

Name	Description
Elements of an ActiveX Control (see page 1408)	An ActiveX control involves many elements which each perform a specific function. The elements include a VCL control, a corresponding COM object wrapper that exposes properties, methods, and events, and one or more associated type libraries.
Designing an ActiveX Control (see page 1409)	When designing an ActiveX control, you start by creating a custom VCL control. This forms the basis of your ActiveX control. For information on creating custom controls, see Creating custom components (see page 1313). When designing the VCL control, keep in mind that it will be embedded in another application; this control is not an application in itself. For this reason, you probably do not want to use elaborate dialog boxes or other major user-interface components. Your goal is typically to make a simple control that works inside of, and follows the rules of the main application. In addition, you should make... more (see page 1409)
Generating an ActiveX Control Based On a VCL Form (see page 1410)	Unlike other ActiveX controls, Active Forms are not first designed and then wrapped by an ActiveX wrapper class. Instead, the ActiveForm wizard generates a blank form that you design later when the wizard leaves you in the Form Designer. When an ActiveForm is deployed on the Web, Delphi creates an HTML page to contain the reference to the ActiveForm and specify its location on the page. The ActiveForm can then displayed and run from a Web browser. Inside the browser, the form behaves just like a stand-alone Delphi form. The form can contain any VCL components or ActiveX controls,... more (see page 1410)

Licensing ActiveX Controls (see page 1410)	Licensing an ActiveX control consists of providing a license key at design-time and supporting the creation of licenses dynamically for controls created at runtime. To provide design-time licenses, a key is created for the control , which is stored in a file with the same name as the project with the LIC extension. This `.LIC` file is added to the project. The user of the control must have a copy of the `.LIC` file to open the control in a development environment. Each control in the project that has Make Control Licensed checked has a separate key entry in... more (see page 1410)
Customizing the ActiveX Control's Interface (see page 1411)	You can add, edit, and remove the properties, methods, and events in an ActiveX control by editing the type library. You can use the Type Library editor as described in Using the Type Library Editor (see page 1453). Remember that when you add events, they should be added to the Events interface, not the ActiveX control's default interface. **Note:** You can add unpublished properties to your ActiveX control's interface. Such properties can be set at runtime and will appear in a development environment, but changes made to them will not persist. That is, when the user of the control changes the value... more (see page 1411)
Adding Additional Properties, Methods, and Events (see page 1412)	You can add additional properties, methods, and events to the control using the type library editor. The declaration is automatically added to the control's implementation unit, type library (TLB) file, and type library unit. The specifics of what Delphi supplies depends on whether you have added a property or method (see page 1412) or whether you have added an event (see page 1413).
How Delphi Adds Properties (see page 1412)	The ActiveX wrapper class implements properties in its interface using read and write access methods. That is, the wrapper class has COM properties, which appear on an interface as getter and/or setter methods. Unlike VCL properties, you do not see a "property" declaration on the interface for COM properties. Rather, you see methods that are flagged as property access methods. When you add a property to the ActiveX control's default interface, the wrapper class definition (which appears in the _TLB unit that is updated by the Type Library editor) gains one or two new methods (a getter and/or setter) that... more (see page 1412)
How Delphi Adds Events (see page 1413)	The ActiveX control can fire events to its container in the same way that an automation object fires events to clients. This mechanism is described in Managing events in your Automation object (see page 1437). If the VCL control you are using as the basis of your ActiveX control has any published events, the wizards automatically add the necessary support for managing a list of client event sinks to your ActiveX wrapper class and define the outgoing dispinterface that clients must implement to respond to events. You add events to this outgoing dispinterface. To add an event in the type library editor,... more (see page 1413)
Enabling Simple Data Binding with the Type Library (see page 1414)	With simple data binding, you can bind a property of your ActiveX control to a field in a database. To do this, the ActiveX control must communicate with its host application about what value represents field data and when it changes. You enable this communication by setting the property's binding flags using the Type Library editor. By marking a property bindable, when a user modifies the property (such as a field in a database), the control notifies its container (the client host application) that the value has changed and requests that the database record be updated. The container interacts with... more (see page 1414)
Creating a Property Page for an ActiveX Control (see page 1415)	A property page is a dialog box similar to the Delphi **Object Inspector** in which users can change the properties of an ActiveX control. A property page dialog allows you to group many properties for a control together to be edited at once. Or, you can provide a dialog box for more complex properties. Typically, users access the property page by right-clicking the ActiveX control and choosing Properties.
Creating a New Property Page (see page 1416)	You use the Property Page wizard to create a new property page.

Adding Controls to a Property Page (see page 1416)	You must add a control to the property page for each property of the ActiveX control that you want the user to access. For example, the following illustration shows a property page for setting the MaskEdit property of an ActiveX control.  The list box allows the user to select from a list of sample masks. The edit controls allow the user to test the mask before applying it to the ActiveX control. You add controls to the property page the same as you would to a form.
Associating Property Page Controls with ActiveX Control Properties (see page 1416)	After adding the controls you need to the property page, you must associate each control with its corresponding property. You make this association by adding code to the property page's UpdatePropertyPage (see page 1416) and UpdateObject (see page 1417) methods.
Updating the Property Page (see page 1416)	Add code to the *UpdatePropertyPage* method to update the control on the property page when the properties of the ActiveX control change. You must add code to the *UpdatePropertyPage* method to update the property page with the current values of the ActiveX control's properties. You can access the ActiveX control using the property page's OleObject property, which is an *OleVariant* that contains the ActiveX control's interface. For example, the following code updates the property page's edit control (InputMask) with the current value of the ActiveX control's *EditMask* property:
Updating the Object (see page 1417)	Add code to the *UpdateObject* method to update the property when the user changes the controls on the property page. You must add code to the *UpdateObject* method in order to set the properties of the ActiveX control to their new values. You use the OleObject property to access the ActiveX control. For example, the following code sets the *EditMask* property of the ActiveX control using the value in the property page's edit box control (InputMask):
Connecting a Property Page to an ActiveX Control (see page 1417)	Describes the steps in connecting an ActiveX control to a property page. **To connect a property page to an ActiveX control:** 1. Add DefinePropertyPage with the GUID constant of the property page as the parameter to the DefinePropertyPages method implementation in the control's implementation for the unit. For example, `procedure TButtonX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage); begin DefinePropertyPage(Class_PropertyPage1); >end; BEGIN_PROPERTY_MAP(TActiveFormXImpl) // Define property pages here. Property pages are defined using // the PROP_PAGE macro with the class id of the page. For example, // PROP_PAGE(CLSID_ActiveFormXPage) PROP_PAGE(CLSID_PropertyPage1) END_PROPERTY_MAP()` The GUID constant, Class_PropertyPage1, of the property page can be found in the property pages unit. The GUID... more (see page 1417)

3.2.2.3.1 Elements of an ActiveX Control

An ActiveX control involves many elements which each perform a specific function. The elements include a VCL control, a corresponding COM object wrapper that exposes properties, methods, and events, and one or more associated type libraries.

VCL control

The underlying implementation of an ActiveX control in Delphi is a VCL control. When you create an ActiveX control, you must first design or choose the VCL control from which you will make your ActiveX control.

The underlying VCL control must be a descendant of *TWinControl*, because it must have a window that can be parented by the host application. When you create an Active form, this object is a descendant of TActiveForm.

Note: The ActiveX control wizard lists the available *TWinControl* descendants from which you can choose to make an ActiveX control. This list does not include all *TWinControl* descendants, however. Some controls, such as *THeaderControl*, are registered as incompatible with ActiveX (using the RegisterNonActiveXprocedure procedure) and do not appear in the list.

ActiveX wrapper

The actual COM object is an ActiveX wrapper object for the VCL control. For Active forms, this class is always TActiveFormControl. For other ActiveX controls, it has a name of the form *TVCLClass*X, where *TVCLClass* is the name of the VCL control class. Thus, for example, the ActiveX wrapper for *TButton* would be named *TButtonX*.

The wrapper class is a descendant of TActiveXControl, which provides support for the ActiveX interfaces. The ActiveX wrapper inherits this support, which allows it to forward Windows messages to the VCL control and parent its window in the host application.

The ActiveX wrapper exposes the VCL control's properties and methods to clients via its default interface. You must implement the wrapper class' properties and methods, delegating method calls to the underlying VCL control. You must also provide the wrapper class with methods that fire the VCL control's events on clients and assign these methods as event handlers on the VCL control.

Type library

You must generate a type library for your ActiveX control that contains the type definitions for the wrapper class, its default interface, and any type definitions that these require. This type information provides a way for your control to advertise its services to host applications. You can view and edit this information using the Type Library editor. Although this information is stored in a separate, binary type library file (.TLB extension), you may also compile it into the ActiveX control DLL as a resource.

Property page

You can optionally give your ActiveX control a property page. The property page allows the user of a host (client) application to view and edit your control's properties. You can group several properties on a page, or use a page to provide a dialog-like interface for a property. For information on how to create property pages, see Creating a property page for an ActiveX control (see page 1415).

3.2.2.3.2 Designing an ActiveX Control

When designing an ActiveX control, you start by creating a custom VCL control. This forms the basis of your ActiveX control. For information on creating custom controls, see Creating custom components (see page 1313).

When designing the VCL control, keep in mind that it will be embedded in another application; this control is not an application in itself. For this reason, you probably do not want to use elaborate dialog boxes or other major user-interface components. Your goal is typically to make a simple control that works inside of, and follows the rules of the main application.

In addition, you should make sure that the types for all properties and methods you want your object to expose to clients are Automation-compatible (see page 1454), because the ActiveX control's interface must support *IDispatch*. The wizards do not add any methods to the wrapper class's interface that have parameters that are not Automation-compatible.

The wizards implement all the necessary ActiveX interfaces required using the COM wrapper class. They also surface all Automation-compatible properties, methods, and events through the wrapper class's default interface. Once a wizard has generated the COM wrapper class and its interface, you can use the Type Library editor (see page 1446) to modify the default interface or augment the wrapper class by implementing additional interfaces.

See Also

Overview of Component Creation (see page 1313)

Elements of an ActiveX Control (see page 1408)

3.2.2.3.3 Generating an ActiveX Control Based On a VCL Form

Unlike other ActiveX controls, Active Forms are not first designed and then wrapped by an ActiveX wrapper class. Instead, the ActiveForm wizard generates a blank form that you design later when the wizard leaves you in the Form Designer.

When an ActiveForm is deployed on the Web, Delphi creates an HTML page to contain the reference to the ActiveForm and specify its location on the page. The ActiveForm can then displayed and run from a Web browser. Inside the browser, the form behaves just like a stand-alone Delphi form. The form can contain any VCL components or ActiveX controls, including custom-built VCL controls.

To start the ActiveForm wizard,

1. Choose **File** ▸ **New** ▸ **Other** to open the New Items dialog box.
2. Select the tab labeled ActiveX.
3. Double-click the ActiveForm icon.

On the Active Form wizard, you can't specify the name of the VCL class to wrap. This is because Active forms are always based on TActiveForm.

You can change the default names for the CoClass, implementation unit, and ActiveX library project. Similarly, this wizard lets you indicate whether you want your Active Form to require a license, whether it should include version information, and whether you want an About box form.

When you exit the wizard, it generates the following:

- An ActiveX Library project file, which contains the code required to start an ActiveX control. You usually don't change this file.
- A type library, which defines and CoClass for your control, the interface it exposes to clients, and any type definitions that these require. For more information about the type library, see Working with type libraries (see page 1445).
- A form that descends from TActiveForm. This form appears in the form designer, where you can use it to visually design the Active Form that appears to clients. Its implementation appears in the generated implementation unit. In the initialization section of the implementation unit, a class factory is created, setting up TActiveFormControl as the ActiveX wrapper for this form.
- An About box form and unit if you requested them.
- A .LIC file if you enabled licensing.

At this point, you can add controls and design the form as you like.

After you have designed and compiled the ActiveForm project into an ActiveX library (which has the OCX extension), you can deploy the project to your Web server and Delphi creates a test HTML page with a reference to the ActiveForm.

See Also

Elements of an ActiveX Control (see page 1408)

Implementing COM objects with Wizards (see page 1397)

Code Generated by Wizards (see page 1398)

3.2.2.3.4 Licensing ActiveX Controls

Licensing an ActiveX control consists of providing a license key at design-time and supporting the creation of licenses dynamically for controls created at runtime.

To provide design-time licenses, a key is created for the control , which is stored in a file with the same name as the project with the LIC extension. This `.LIC` file is added to the project. The user of the control must have a copy of the `.LIC` file to open the control in a development environment. Each control in the project that has Make Control Licensed checked has a separate key entry in the `.LIC` file.

To support runtime licenses, the wrapper class implements two methods, *GetLicenseString* and *GetLicenseFilename*. These return the license string for the control and the name of the `.LIC` file, respectively. When a host application tries to create the ActiveX control, the class factory for the control calls these methods and compares the string returned by *GetLicenseString* with the string stored in the `.LIC` file.

Runtime licenses for the Internet Explorer require an extra level of indirection because users can view HTML source code for any Web page, and because an ActiveX control is copied to the user's computer before it is displayed. To create runtime licenses for controls used in Internet Explorer, you must first generate a license package file (LPK file) and embed this file in the HTML page that contains the control. The LPK file is essentially an array of ActiveX control CLSIDs and license keys.

Note: To generate the LPK file, use the utility, LPK_TOOL.EXE, which you can download from the Microsoft Web site (www.microsoft.com).

To embed the LPK file in a Web page, use the HTML objects, <OBJECT> and <PARAM> as follows:

```
<OBJECT CLASSID="clsid:6980CB99-f75D-84cf-B254-55CA55A69452">
<PARAM NAME="LPKPath" VALUE="ctrllic.lpk">
</OBJECT>
```

The CLSID identifies the object as a license package and PARAM specifies the relative location of the license package file with respect to the HTML page.

When Internet Explorer tries to display the Web page containing the control, it parses the LPK file, extracts the license key, and if the license key matches the control's license (returned by *GetLicenseString*), it renders the control on the page. If more than one LPK is included in a Web page, Internet Explorer ignores all but the first.

For more information, look for Licensing ActiveX Controls on the Microsoft Web site.

3.2.2.3.5 Customizing the ActiveX Control's Interface

You can add, edit, and remove the properties, methods, and events in an ActiveX control by editing the type library. You can use the Type Library editor as described in Using the Type Library Editor (see page 1453). Remember that when you add events, they should be added to the Events interface, not the ActiveX control's default interface.

Note: You can add unpublished properties to your ActiveX control's interface. Such properties can be set at runtime and will appear in a development environment, but changes made to them will not persist. That is, when the user of the control changes the value of a property at design time, the changes are not reflected when the control is run. If the source is a VCL object and the property is not already published, you can make properties persistent by creating a descendant of the VCL object and publishing the property in the descendant.

You may also choose not to expose all of the VCL control's properties, methods, and events to host applications. You can use the Type Library editor to remove these from the interfaces that the wizard generated. When you remove properties and methods from an interface using the Type Library editor, the Type Library editor does not remove them from the corresponding implementation class. Edit the ActiveX wrapper class in the implementation unit to remove these after you have changed the interface in the Type Library editor.

Warning: Any changes you make to the type library will be lost if you regenerate the ActiveX control from the original VCL control or form.

Tip: It is a good idea to check the methods that the wizard adds to your ActiveX wrapper class. Not only does this give you a chance to note where the wizard omitted any data-aware properties or methods that were not Automation-compatible, it also lets you detect methods for which the wizard could not generate an implementation. Such methods appear with a comment in the

implementation that indicates the problem.

See Also

Generating an ActiveX Control Based On a VCL Form (see page 1410)

3.2.2.3.6 Adding Additional Properties, Methods, and Events

You can add additional properties, methods, and events to the control using the type library editor. The declaration is automatically added to the control's implementation unit, type library (TLB) file, and type library unit. The specifics of what Delphi supplies depends on whether you have added a property or method (see page 1412) or whether you have added an event (see page 1413).

See Also

Customizing the ActiveX Control's Interface (see page 1411)

3.2.2.3.7 How Delphi Adds Properties

The ActiveX wrapper class implements properties in its interface using read and write access methods. That is, the wrapper class has COM properties, which appear on an interface as getter and/or setter methods. Unlike VCL properties, you do not see a "property" declaration on the interface for COM properties. Rather, you see methods that are flagged as property access methods. When you add a property to the ActiveX control's default interface, the wrapper class definition (which appears in the _TLB unit that is updated by the Type Library editor) gains one or two new methods (a getter and/or setter) that you must implement, just as when you add a method to the interface, the wrapper class gains a corresponding method for you to implement. Thus, adding properties to the wrapper class's interface is essentially the same as adding methods: the wrapper class definition gains new skeletal method implementations for you to complete.

Note: For details on what appears in the generated _TLB unit, see Code generated when you import type library information (see page 1421).

For example, consider a *Caption* property, of type TCaption in the underlying VCL object. To Add this property to the object's interface, you enter the following when you add a property to the interface via the type library editor:

```
property Caption: TCaption read Get_Caption write Set_Caption;
```

Delphi adds the following declarations to the wrapper class:

```
function Get_Caption: WideString; safecall;
procedure Set_Caption(const Value: WideString); safecall;
STDMETHOD(get_Caption(BSTR* Value));
STDMETHOD(set_Caption(BSTR Value));
```

In addition, it adds skeletal method implementations for you to complete:

```
function TButtonX.Get_Caption: WideString;
begin
end;
procedure TButtonX.Set_Caption(Value: WideString);
begin
end;
STDMETHODIMP TButtonXImpl::get_Caption(BSTR* Value)
{
  try
  {
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
```

```
};
STDMETHODIMP TButtonXImpl::set_Caption(BSTR Value)
{
  try
  {
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
```

Typically, you can implement these methods by simply delegating to the associated VCL control, which can be accessed using the *FDelphiControl* member of the wrapper class:

```
function TButtonX.Get_Caption: WideString;
begin
   Result := WideString(FDelphiControl.Caption);
end;

procedure TButtonX.Set_Caption(const Value: WideString);
begin
   FDelphiControl.Caption := TCaption(Value);
end;
STDMETHODIMP TButtonXImpl::get_Caption(BSTR* Value)
{
try
{
*Value = WideString(m_VclCtl->Caption).Copy();
}
catch(Exception &e)
{
return Error(e.Message.c_str(), IID_IButtonX);
}
return S_OK;
};
STDMETHODIMP TButtonXImpl::set_Caption(BSTR Value)
{
  try
  {
    m_VclCtl->Caption = AnsiString(Value);
  }
  catch(Exception &e)
  {
    return Error(e.Message.c_str(), IID_IButtonX);
  }
  return S_OK;
};
```

In some cases, you may need to add code to convert the COM data types to native Delphi types. The preceding example manages this with typecasting.

Note: Because the Automation interface methods are declared**safecall, you do not have to implement COM exception code for these methods—the Delphi compiler handles this for you by generating code around the body of safecall methods to catch Delphi exceptions and to convert them into COM error info structures and return codes.**

See Also

Code Generated When You Import Type Library Information (see page 1421)

3.2.2.3.8 How Delphi Adds Events

The ActiveX control can fire events to its container in the same way that an automation object fires events to clients. This

mechanism is described in Managing events in your Automation object (see page 1437).

If the VCL control you are using as the basis of your ActiveX control has any published events, the wizards automatically add the necessary support for managing a list of client event sinks to your ActiveX wrapper class and define the outgoing dispinterface that clients must implement to respond to events.

You add events to this outgoing dispinterface. To add an event in the type library editor, select the event interface and click on the method icon. Then manually add the list of parameters you want include using the parameter page.

Next, you must declare a method in your wrapper class that is of the same type as the event handler for the event in the underlying VCL control. This is not generated automatically, because Delphi does not know which event handler you are using:

```
procedure KeyPressEvent(Sender: TObject; var Key: Char);
```

Implement this method to use the host application's event sink, which is stored in the wrapper class's *FEvents* member:

```
procedure TButtonX.KeyPressEvent(Sender: TObject; var Key: Char);
var
  TempKey: Smallint;
begin
    TempKey := Smallint(Key); {cast to an OleAutomation compatible type }
    if FEvents <> nil then
      FEvents.OnKeyPress(TempKey)
    Key := Char(TempKey);
  end;
void __fastcall TButtonXImpl::KeyPressEvent(TObject *Sender, char &Key)
{
short TempKey;
TempKey = (short)Key;
Fire_OnKeyPress(&TempKey);
Key = (short)TempKey;
};
```

Note: When firing events in an ActiveX control, you do not need to iterate through a list of event sinks because the control only has a single host application. This is simpler than the process for most Automation servers.

Finally, you must assign this event handler to the underlying VCL control, so that it is called when the event occurs. You make this assignment in the *InitializeControl* method:

```
procedure TButtonX.InitializeControl;
begin
  FDelphiControl := Control as TButton;
  FDelphiControl.OnClick := ClickEvent;
  FDelphiControl.OnKeyPress := KeyPressEvent;
end;
void InitializeControl()
{
  m_VclCtl->OnClick = ClickEvent;
  m_VclCtl->OnKeyPress = KeyPressEvent;
}
```

See Also

Code Generated When You Import Type Library Information (see page 1421)

3.2.2.3.9 Enabling Simple Data Binding with the Type Library

With simple data binding, you can bind a property of your ActiveX control to a field in a database. To do this, the ActiveX control must communicate with its host application about what value represents field data and when it changes. You enable this communication by setting the property's binding flags using the Type Library editor.

By marking a property bindable, when a user modifies the property (such as a field in a database), the control notifies its container (the client host application) that the value has changed and requests that the database record be updated. The

container interacts with the database and then notifies the control whether it succeeded or failed to update the record.

Note: The container application that hosts your ActiveX control is responsible for connecting the data-aware properties you enable in the type library to the database.

Use the type library to enable simple data binding,

1. On the toolbar, click the property that you want to bind.
2. Choose the flags page.
3. Select the following binding attributes:

Binding attribute	Description
Bindable	Indicates that the property supports data binding. If marked bindable, the property notifies its container when the property value has changed.
Request Edit	Indicates that the property supports the OnRequestEdit notification. This allows the control to ask the container if its value can be edited by the user.
Display Bindable	Indicates that the container can show users that this property is bindable.
Default Bindable	Indicates the single, bindable property that best represents the object. Properties that have the default bind attribute must also have the bindable attribute. Cannot be specified on more than one property in a dispinterface.
Immediate Bindable	Allows individual bindable properties on a form to specify this behavior. When this bit is set, all changes will be notified. The bindable and request edit attribute bits need to be set for this new bit to have an effect.

4. Click the Refresh button on the toolbar to update the type library. To test a data-binding control, you must register it first. For example, to convert a *TEdit* control into a data-bound ActiveX control, create the ActiveX control from a *TEdit* and then change the Text property flags to Bindable, Display Bindable, Default Bindable, and Immediate Bindable. After the control is registered and imported, it can be used to display data.

See Also

Working with Type Libraries: Overview (see page 1445)

3.2.2.3.10 Creating a Property Page for an ActiveX Control

A property page is a dialog box similar to the Delphi **Object Inspector** in which users can change the properties of an ActiveX control. A property page dialog allows you to group many properties for a control together to be edited at once. Or, you can provide a dialog box for more complex properties.

Typically, users access the property page by right-clicking the ActiveX control and choosing Properties.

The process of creating a property page is similar to creating a form, you

1. Create a new property page (see page 1416).
2. Add controls to the property page (see page 1416).
3. Associate the controls the property page with the properties of an ActiveX control (see page 1416).
4. Connect the property page to the ActiveX control (see page 1417).

   Note: When adding properties to an ActiveX control or ActiveForm, you must publish the properties that you want to persist. If they are not published in the underlying VCL control, you must make a custom descendant of the VCL control that redeclares the properties as published and then create an ActiveX control from the descendant class.

3.2.2.3.11 Creating a New Property Page

You use the Property Page wizard to create a new property page.

To create a new property page,

1. Choose **File** ▶ **New** ▶ **Other**.
2. Select the ActiveX folder under Delphi Projects .
3. Double-click the Property Page icon in the right pane.

The wizard creates a new form and implementation unit for the property page. The form is a descendant of TPropertyPage, which lets you associate the form with the ActiveX control whose properties it edits.

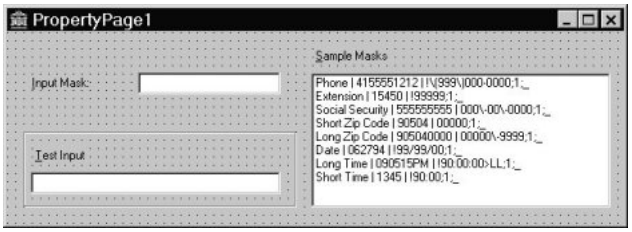
See Also

Creating a Property Page for an ActiveX Control (see page 1415)

3.2.2.3.12 Adding Controls to a Property Page

You must add a control to the property page for each property of the ActiveX control that you want the user to access.

For example, the following illustration shows a property page for setting the MaskEdit property of an ActiveX control.



The list box allows the user to select from a list of sample masks. The edit controls allow the user to test the mask before applying it to the ActiveX control. You add controls to the property page the same as you would to a form.

See Also

Creating a Property Page for an ActiveX Control (see page 1415)

3.2.2.3.13 Associating Property Page Controls with ActiveX Control Properties

After adding the controls you need to the property page, you must associate each control with its corresponding property. You make this association by adding code to the property page's UpdatePropertyPage (see page 1416) and UpdateObject (see page 1417) methods.

See Also

Creating a Property Page for an ActiveX Control (see page 1415)

3.2.2.3.14 Updating the Property Page

Add code to the *UpdatePropertyPage* method to update the control on the property page when the properties of the ActiveX control change. You must add code to the *UpdatePropertyPage* method to update the property page with the current values of the ActiveX control's properties.

You can access the ActiveX control using the property page's OleObject property, which is an *OleVariant* that contains the ActiveX control's interface.

For example, the following code updates the property page's edit control (InputMask) with the current value of the ActiveX control's *EditMask* property:

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
   { Update your controls from OleObject }
   InputMask.Text := OleObject.EditMask;
end;
```

For example, the following code updates the property page's edit control (InputMask) with tthe current value of the ActiveX control's EditMask property:

```
void __fastcall TPropertyPage1::UpdatePropertyPage(void)
{
InputMask->Text = OleObject.OlePropertyGet("EditMask");
}
```

Note: It is also possible to write a property page that represents more than one ActiveX control. In this case, you don't use the *OleObject* property. Instead, you must iterate through a list of interfaces that is maintained by the *OleObjects* property.

See Also

Creating a Property Page for an ActiveX Control (see page 1415)

3.2.2.3.15 Updating the Object

Add code to the *UpdateObject* method to update the property when the user changes the controls on the property page. You must add code to the *UpdateObject* method in order to set the properties of the ActiveX control to their new values.

You use the OleObject property to access the ActiveX control.

For example, the following code sets the *EditMask* property of the ActiveX control using the value in the property page's edit box control (InputMask):

```
procedure TPropertyPage1.UpdateObject;
begin
   {Update OleObject from your control }
   OleObject.EditMask := InputMask.Text;
end;
void __fastcall TPropertyPage1::UpdateObject(void)
{
  // Update OleObject from your control
  OleObject.OlePropertySet<WideString>("EditMask", WideString(InputMast->Text).Copy());
}
```

See Also

Creating a Property Page for an ActiveX Control (see page 1415)

3.2.2.3.16 Connecting a Property Page to an ActiveX Control

Describes the steps in connecting an ActiveX control to a property page.

To connect a property page to an ActiveX control:

1. Add DefinePropertyPage with the GUID constant of the property page as the parameter to the DefinePropertyPages method implementation in the control's implementation for the unit. For example, `procedure TButtonX.DefinePropertyPages(DefinePropertyPage: TDefinePropertyPage); begin DefinePropertyPage(Class_PropertyPage1); >end; BEGIN_PROPERTY_MAP(TActiveFormXImpl) // Define property pages here. Property pages are defined using // the PROP_PAGE macro with the class id of the page. For example, // PROP_PAGE(CLSID_ActiveFormXPage) PROP_PAGE(CLSID_PropertyPage1) END_PROPERTY_MAP()` The GUID constant, Class_PropertyPage1, of the property page can be found in the property pages unit. The GUID is defined in the property page's implementation unit .

2. Add the property page unit to the **uses** clause of the controls implementation unit.

See Also

Creating a Property Page for an ActiveX Control (see page 1415)

3.2.2.4 Creating COM clients

Topics

Name	Description
Creating COM Clients (see page 1419)	COM clients are applications that make use of a COM object implemented by another application or library. The most common types are applications that control an Automation server (see page 1430) (Automation controllers) and applications that host an ActiveX control (ActiveX containers). At first glance these two types of COM client are very different: The typical Automation controller launches an external server EXE and issues commands to make that server perform tasks on its behalf. The Automation server is usually nonvisual and out-of-process. The typical ActiveX client, on the other hand, hosts a visual control, using it much the same way you use... more (see page 1419)
Importing Type Library Information (see page 1420)	To make information about the COM server available to your client application, you must import the information about the server that is stored in the server's type library (see page 1395). Your application can then use the resulting generated classes to control the server object. There are two ways to import type library information: • You can use the **Import Component dialog** to import all available information about the server types, objects, and interfaces. This is the most general method, because it lets you import information from any type library and can optionally generate component wrappers for all creatable CoClasses in the type... more (see page 1420)
Code Generated When You Import Type Library Information (see page 1421)	Once you import a type library, you can view the generated *TypeLibName*_TLB unit. At the top, you will find the following: First, constant declarations giving symbolic names to the GUIDS of the type library and its interfaces and CoClasses. The names for these constants are generated as follows: • the GUID for the type library has the form LBID_*TypeLibName*, where *TypeLibName* is the name of the type library. • The GUID for an interface has the form IID_*InterfaceName*, where *InterfaceName* is the name of the interface. • The GUID for a dispinterface has the form DIID_*InterfaceName*,... more (see page 1421)
Controlling an Imported Object (see page 1422)	After importing type library information (see page 1420), you are ready to start programming with the imported objects. How you proceed depends in part on the objects, and in part on whether you have chosen to create component wrappers. There are two basic approaches: • Using component wrappers (see page 1422). • Writing client code based on type library definitions (see page 1423).

Using Component Wrappers (see page 1422)	If you generated a component wrapper for your server object, writing your COM client application is not very different from writing any other application that contains VCL components. The server object's properties, methods, and events are already encapsulated in the VCL component. You need only assign event handlers, set property values, and call methods. To use the properties, methods, and events of the server object, see the documentation for your server. The component wrapper automatically provides a dual interface (see page 1438) where possible. Delphi determines the VTable layout from information in the type library. In addition, your new component inherits certain important... more (see page 1422)
Writing Client Code Based On Type Library Definitions (see page 1423)	Although you must use a component wrapper for hosting an ActiveX control, you can write an Automation controller using only the definitions from the type library that appear in the *TypeLibName*_TLB unit. This process is a bit more involved that letting a component do the work, especially if you need to respond to events. The following topics describe how to implement the various actions your Automation controller needs to perform: • Connect to the server (see page 1424). • Control the Automation server using a dual interface (see page 1424). • Control the Automation server using a dispinterface (see page 1424). • Respond to events generated by the Automation (see page 1425)... more (see page 1423)
Connecting to a Server (see page 1424)	Before you can drive an Automation server from your controller application, you must obtain a reference to an interface it supports. Typically, you connect to a server through its main interface. If the main interface is a dual interface (see page 1438), you can use the creator objects in the *TypeLibName*_TLB.pas file. The creator classes have the same name as the CoClass, with the prefix "Co" added. You can connect to a server on the same machine by calling the *Create* method, or a server on a remote machine using the *CreateRemote* method. Because *Create* and *CreateRemote* are class methods, you... more (see page 1424)
Controlling an Automation Server Using a Dual Interface (see page 1424)	After using the automatically generated creator class to connect to the server (see page 1424), you call methods of the interface. For example,
Controlling an Automation Server Using a Dispatch Interface (see page 1424)	Typically, you use the dual interface (see page 1424) to control the Automation server. However, you may find a need to control an Automation server with a dispatch interface (see page 1438) because no dual interface is available.
Handling Events in an Automation Controller (see page 1425)	When you generate a Component wrapper for an object whose type library you import, you can respond to events simply using the events that are added to the generated component. If you do not use a Component wrapper, however, (or if the server uses COM+ events), you must write the event sink code yourself.
Creating Clients for Servers That Do Not Have a Type Library (see page 1426)	Some older COM technologies, such as object linking and embedding (OLE), do not provide type information in a type library. Instead, they rely on a standard set of predefined interfaces. To write clients that host such objects, you can use the TOleContainer component. This component appears on the System category of the **Tool Palette.** *TOleContainer* acts as a host site for an Ole2 object. It implements the *IOleClientSite* interface and, optionally, *IOleDocumentSite*. Communication is handled using OLE verbs.

3.2.2.4.1 Creating COM Clients

COM clients are applications that make use of a COM object implemented by another application or library. The most common types are applications that control an Automation server (see page 1430) (Automation controllers) and applications that host an ActiveX control (ActiveX containers).

At first glance these two types of COM client are very different: The typical Automation controller launches an external server EXE and issues commands to make that server perform tasks on its behalf. The Automation server is usually nonvisual and out-of-process. The typical ActiveX client, on the other hand, hosts a visual control, using it much the same way you use any control on the Component palette. ActiveX servers are always in-process servers.

However, the task of writing these two types of COM client is remarkably similar: The client application obtains an interface for the server object and uses its properties and methods. RAD Studio makes this particularly easy by letting you wrap the server CoClass in a component on the client, which you can even install on the Component palette.

When writing a COM client, you must understand the interface that the server exposes to clients, just as you must understand the properties and methods of a component from the Component palette to use it in your application. This interface (or set of interfaces) is determined by the server application, and typically published in a type library (see page 1445). For specific information on a particular server application's published interfaces, you should consult that application's documentation.

Even if you do not choose to wrap a server object in a component wrapper and install it on the Component palette, you must make its interface definition available to your application. To do this, you can import the server's type information (see page 1420).

Once you have imported the type information, you can write code to control the imported object (see page 1422).

Note: You can also query the type information directly using COM APIs, but RAD Studio provides no special support for this.

Some older COM technologies, such as object linking and embedding (OLE), do not provide type information in a type library. Instead, they rely on a standard set of predefined interfaces. These are discussed in Creating Clients for Servers That Do Not Have a Type Library (see page 1426).

See Also

Creating Active Server Pages (see page 1400)

Working with Type Libraries: Overview (see page 1445)

Overview of COM Technologies (see page 1385)

3.2.2.4.2 Importing Type Library Information

To make information about the COM server available to your client application, you must import the information about the server that is stored in the server's type library (see page 1395). Your application can then use the resulting generated classes to control the server object.

There are two ways to import type library information:

- You can use the **Import Component dialog** to import all available information about the server types, objects, and interfaces. This is the most general method, because it lets you import information from any type library and can optionally generate component wrappers for all creatable CoClasses in the type library that are not flagged as Hidden, Restricted, or PreDeclID.
- You can also use the **Import Component dialog** if you are importing from the type library of an ActiveX control. This imports the same type information, but only creates component wrappers for CoClasses that represent ActiveX controls.
- You can use the command line utility tlibimp.exe which provides additional configuration options not available from within the IDE.
- A type library generated using a wizard is automatically imported using the same mechanism as the import type library menu item.

Regardless of which method you choose to import type library information, the resulting dialog creates a unit with the name *TypeLibName*_TLB, where *TypeLibName* is the name of the type library. This file contains declarations for the classes, types, and interfaces defined in the type library. By including it in your project, those definitions are available to your application so that you can create objects and call their interfaces. This file may be recreated by the IDE from time to time; as a result, making manual changes to the file is not recommended.

In addition to adding type definitions to the *TypeLibName*_TLB unit, the dialog can also create VCL class wrappers for any CoClasses defined in the type library. When you use the Import Type Library dialog, these wrappers are optional. When you use the Import ActiveX dialog, they are always generated for all CoClasses that represent controls.

The generated class wrappers represent the CoClasses to your application, and expose the properties and methods of its interfaces. If a CoClass supports the interfaces for generating events (*IConnectionPointContainer* and *IConnectionPoint*), the VCL class wrapper creates an event sink so that you can assign event handlers for the events as simply as you can for any

other component. If you tell the dialog to install the generated VCL classes on the **Tool Palette**, you can use the **Object Inspector** to assign property values and event handlers.

Note: The Import Component dialog

does not create class wrappers for COM+ event objects. To write a client that responds to events generated by a COM+ event object, you must create the event sink programmatically. This process is described in Handling COM+ events (see page 1425). For more details about the code generated when you import a type library, see Code generated when you import type library information (see page 1421).

See Also

Working with Type Libraries (see page 1445)

Controlling an Imported Object (see page 1422)

3.2.2.4.3 Code Generated When You Import Type Library Information

Once you import a type library, you can view the generated *TypeLibName*_TLB unit. At the top, you will find the following:

First, constant declarations giving symbolic names to the GUIDS of the type library and its interfaces and CoClasses. The names for these constants are generated as follows:

- the GUID for the type library has the form LBID_*TypeLibName*, where *TypeLibName* is the name of the type library.
- The GUID for an interface has the form IID_*InterfaceName*, where *InterfaceName* is the name of the interface.
- The GUID for a dispinterface has the form DIID_*InterfaceName*, where *InterfaceName* is the name of the dispinterface.
- The GUID for a CoClass has the form CLASS_*ClassName*, where *ClassName* is the name of the CoClass.
- The compiler directive VARPROPSETTER will be on. This allows the use of the keyword var in the parameter list of property setter methods. This disables a compiler optimization that would cause parameters to be passed by value instead of by reference. The VARPROPSETTER directive must be on, when creating TLB units for components written in a language other than Delphi.

Second, declarations for the CoClasses in the type library. These map each CoClass to its default interface.

Third, declarations for the interfaces and dispinterfaces in the type library.

Fourth, declarations for a creator class for each CoClass whose default interface supports VTable binding. The creator class has two class methods, *Create* and *CreateRemote*, that can be used to instantiate the CoClass locally (*Create*) or remotely (*CreateRemote*).These methods return the default interface for the CoClass.

These declarations provide you with what you need to create instances of the CoClass and access its interface. All you need do is add the generated *TypeLibName*_TLB.pas file to the uses clause of the unit where you wish to bind to a CoClass and call its interfaces.

Note: This portion of the *TypeLibName*_TLB unit is also generated when you use the Type Library editor or the command-line utility TLIBIMP.

If you want to use an ActiveX control, you also need the generated VCL wrapper in addition to the declarations described above. The VCL wrapper handles window management issues for the control. You may also have generated a VCL wrapper for other CoClasses in the Import Type Library dialog. These VCL wrappers simplify the task of creating server objects and calling their methods. They are especially recommended if you want your client application to respond to events.

The declarations for generated VCL wrappers appear at the bottom of the interface section. Component wrappers for ActiveX controls are descendants of TOleControl. Component wrappers for Automation objects descend from TOleServer. The generated component wrapper adds the properties, events, and methods exposed by the CoClass's interface. You can use this component like any other VCL component.

Warning: You should not edit the generated *TypeLibName*_TLB unit. It is regenerated each time the type library is refreshed, so any changes will be overwritten.

Note: For the most up-to-date information about the generated code, refer to the comments in the automatically-generated *TypeLibName*_TLB unit.

3.2.2.4.4 Controlling an Imported Object

After importing type library information (see page 1420), you are ready to start programming with the imported objects. How you proceed depends in part on the objects, and in part on whether you have chosen to create component wrappers. There are two basic approaches:

- Using component wrappers (see page 1422).
- Writing client code based on type library definitions (see page 1423).

See Also

Importing Type Library Information (see page 1420)

3.2.2.4.5 Using Component Wrappers

If you generated a component wrapper for your server object, writing your COM client application is not very different from writing any other application that contains VCL components. The server object's properties, methods, and events are already encapsulated in the VCL component. You need only assign event handlers, set property values, and call methods.

To use the properties, methods, and events of the server object, see the documentation for your server. The component wrapper automatically provides a dual interface (see page 1438) where possible. Delphi determines the VTable layout from information in the type library.

In addition, your new component inherits certain important properties and methods from its base class.

ActiveX wrappers

You should always use a component wrapper when hosting ActiveX controls, because the component wrapper integrates the control's window into the VCL framework.

The properties and methods an ActiveX control inherits from TOleControl allow you to access the underlying interface or obtain information about the control. Most applications, however, do not need to use these. Instead, you use the imported control the same way you would use any other VCL control.

Typically, ActiveX controls provide a property page that lets you set their properties. Property pages are similar to the component editors some components display when you double-click on them in the form designer. To display an ActiveX control's property page, right click and choose Properties.

The way you use most imported ActiveX controls is determined by the server application. However, ActiveX controls use a standard set of notifications when they represent the data from a database field. See TOleControl for information on how to host such ActiveX controls.

Automation object wrappers

The wrappers for Automation objects let you control how you want to form the connection to your server object:

First, the ConnectKind property indicates whether the server is local or remote and whether you want to connect to a server that is already running or if a new instance should be launched. When connecting to a remote server, you must specify the machine name using the RemoteMachineName property.

Second, once you have specified the *ConnectKind*, there are three ways you can connect your component to the server:

- you can explicitly connect to the server by calling the component's Connect method.
- You can tell the component to connect automatically when your application starts up by setting the AutoConnect property to *true*.
- You do not need to explicitly connect to the server. The component automatically forms a connection when you use one of the server's properties or methods using the component.

Calling methods or accessing properties is the same as using any other component:

```
TServerComponent1.DoSomething;
TServerComponent1->DoSomething();
```

Handling events is easy, because you can use the **Object Inspector** to write event handlers. Note, however, that the event handler on your component may have slightly different parameters than those defined for the event in the type library. Specifically, pointer types (var parameters and interface pointers) are changed to Variants. You must explicitly cast var parameters to the underlying type before assigning a value. Interface pointers can be cast to the appropriate interface type using the **as** operator.

For example, the following code shows an event handler for the ExcelApplication event, OnNewWorkBook. The event handler has a parameter that provides the interface of another CoClass (ExcelWorkbook). However, the interface is not passed as an ExcelWorkbook interface pointer, but rather as an OleVariant.

```
procedure TForm1.XLappNewWorkbook(Sender: TObject; var Wb:OleVariant);
begin
  { Note how the OleVariant for the interface must be cast to the correct type }
  ExcelWorkbook1.ConnectTo((iUnknown(wb) as ExcelWorkbook));
end;
void _fastcall TForm1::XLappNewWorkbook(TObject *Sender, ExcelWorkbookPtr Wb)
{
ExcelWorkbook1->ConnectTo(Wb);
}
```

In this example, the event handler assigns the workbook to an ExcelWorkbook component (ExcelWorkbook1). This demonstrates how to connect a component wrapper to an existing interface by using the *ConnectTo* method. The *ConnectTo* method is added to the generated code for the component wrapper.

Servers that have an application object expose a Quit method on that object to let clients terminate the connection. Quit typically exposes functionality that is equivalent to using the File menu to quit the application. Code to call the Quit method is generated in your component's Disconnect method. If it is possible to call the Quit method with no parameters, the component wrapper also has an *AutoQuit* property. *AutoQuit* causes your controller to call Quit when the component is freed. If you want to disconnect at some other time, or if the Quit method requires parameters, you must call it explicitly. Quit appears as a public method on the generated component.

See Also

Writing Client Code Based On Type Library Definitions (see page 1423)

Creating Clients for Servers That Do Not Have a Type Library (see page 1426)

3.2.2.4.6 Writing Client Code Based On Type Library Definitions

Although you must use a component wrapper for hosting an ActiveX control, you can write an Automation controller using only the definitions from the type library that appear in the *TypeLibName*_TLB unit. This process is a bit more involved that letting a component do the work, especially if you need to respond to events.

The following topics describe how to implement the various actions your Automation controller needs to perform:

- Connect to the server (see page 1424).
- Control the Automation server using a dual interface (see page 1424).
- Control the Automation server using a dispinterface (see page 1424).
- Respond to events generated by the Automation server (see page 1425).

See Also

Using Component Wrappers (see page 1422)

Creating Clients for Servers That Do Not Have a Type Library (see page 1426)

3.2.2.4.7 Connecting to a Server

Before you can drive an Automation server from your controller application, you must obtain a reference to an interface it supports. Typically, you connect to a server through its main interface.

If the main interface is a dual interface (see page 1438), you can use the creator objects in the *TypeLibName*_TLB.pas file. The creator classes have the same name as the CoClass, with the prefix "Co" added. You can connect to a server on the same machine by calling the *Create* method, or a server on a remote machine using the *CreateRemote* method. Because *Create* and *CreateRemote* are class methods, you do not need an instance of the creator class to call them.

```
MyInterface := CoServerClassName.Create;
MyInterface := CoServerClassName.CreateRemote('Machine1');
pInterface = CoServerClassName.Create();
pInterface = CoServerClassName.CreateRemote("Machine1");
```

Create and *CreateRemote* return the default interface for the CoClass.

If the default interface is a dispatch interface, then there is no Creator class generated for the CoClass. Instead, you can call the global CreateOleObject function, passing in the GUID for the CoClass (there is a constant for this GUID defined at the top of the _TLB unit). *CreateOleObject* returns an IDispatch pointer for the default interface.

See Also

Controlling an Automation Server Using a Dispatch Interface (see page 1424)

Controlling an Automation Server Using a Dual Interface (see page 1424)

Handling Events in an Automation Controller (see page 1425)

3.2.2.4.8 Controlling an Automation Server Using a Dual Interface

After using the automatically generated creator class to connect to the server (see page 1424), you call methods of the interface. For example,

```
var
  MyInterface : _Application;
begin
  MyInterface := CoWordApplication.Create;
  MyInterface.DoSomething;
TComApplication AppPtr = CoWordApplication_.Create();
AppPtr->DoSomething;
```

The interface and creator class are defined in the *TypeLibName*_TLB unit that is generated automatically when you import a type library.

See Also

Controlling an Automation Server Using a Dispatch Interface (see page 1424)

3.2.2.4.9 Controlling an Automation Server Using a Dispatch Interface

Typically, you use the dual interface (see page 1424) to control the Automation server. However, you may find a need to control an Automation server with a dispatch interface (see page 1438) because no dual interface is available.

To call the methods of a dispatch interface,

1. Connect to the server (see page 1424), using the global CreateOleObject function.
2. Use the **as** operator to cast the *IDispatch* interface returned by *CreateOleObject* to the dispinterface for the CoClass. This dispinterface type is declared in the *TypeLibName*_TLB unit.

3. Control the Automation server by calling methods of the dispinterface. Another way to use dispatch interfaces is to assign them to a *Variant*. By assigning the interface returned by *CreateOleObject* to a *Variant*, you can take advantage of the *Variant* type's built-in support for interfaces. Simply call the methods of the interface, and the *Variant* automatically handles all *IDispatch* calls, fetching the dispatch ID and invoking the appropriate method. The Variant type includes built-in support for calling dispatch interfaces, through its **var.**

```
  V: Variant;
begin
V:= CreateOleObject("TheServerObject");
V.MethodName; { calls the specified method }
...
```

An advantage of using *Variants* is that you do not need to import the type library, because *Variants* use only the standard *IDispatch* methods to call the server. The trade-off is that *Variants* are slower, because they use dynamic binding at runtime.

See Also

Controlling an Automation Server Using a Dual Interface (see page 1424)

3.2.2.4.10 Handling Events in an Automation Controller

When you generate a Component wrapper for an object whose type library you import, you can respond to events simply using the events that are added to the generated component. If you do not use a Component wrapper, however, (or if the server uses COM+ events), you must write the event sink code yourself.

Handling Automation events programmatically

Before you can handle events, you must define an event sink. This is a class that implements the event dispatch interface that is defined in the server's type library.

To write the event sink, create an object that implements the event dispatch interface:

```
TServerEventsSink = class(TObject, _TheServerEvents)
...{ declare the methods of _TheServerEvents here }
end;
class MyEventSinkClass: TEventDispatcher<MyEventSinkClass, DIID_TheServerEvents>
{
...// declare the methods of DIID_TheServerEvents here
}
```

Once you have an instance of your event sink, you must inform the server object of its existence so that the server can call it. To do this, you call the global InterfaceConnect procedure, passing it

- The interface to the server that generates events.
- The GUID for the event interface that your event sink handles.
- An IUnknown interface for your event sink.
- A variable that receives a Longint that represents the connection between the server and your event sink.

```
{MyInterface is the server interface you got when you connected to the server }
InterfaceConnect(MyInterface, DIID_TheServerEvents,
                 MyEventSinkObject as IUnknown, cookievar);
pInterface = CoServerClassName.CreateRemote("Machine1");
MyEventSinkClass ES;
ES.ConnectEvents(pInterface);
```

After calling *InterfaceConnect*, your event sink is connected and receives calls from the server when events occur.

You must terminate the connection before you free your event sink. To do this, call the global InterfaceDisconnect procedure, passing it all the same parameters except for the interface to your event sink (and the final parameter is ingoing rather than outgoing):

```
InterfaceDisconnect(MyInterface, DIID_TheServerEvents, cookievar);
ES.DisconnectEvents(pInterface);
```

Note: You must be certain that the server has released its connection to your event sink before you free it. Because you don't know how the server responds to the disconnect notification initiated by *InterfaceDisconnect*, this may lead to a race condition if you free your event sink immediately after the call. The easiest way to guard against problems is to have your event sink maintain its own reference count that is not decremented until the server releases the event sink's interface.

Handling COM+ events

Under COM+, servers use a special helper object to generate events rather than a set of special interfaces (*IConnectionPointContainer* and *IConnectionPoint*). Because of this, you can't use an event sink that descends from *TEventDispatcher. TEventDispatcher* is designed to work with those interfaces, not COM+ event objects.

Instead of defining an event sink, your client application defines a subscriber object. Subscriber objects, like event sinks, provide the implementation of the event interface. They differ from event sinks in that they subscribe to a particular event object rather than connecting to a server's connection point.

To define a subscriber object, use the COM Object wizard (see page 1431), selecting the event object's interface as the one you want to implement. The wizard generates an implementation unit with skeletal methods that you can fill in to create your event handlers.

Note: You may need to add the event object's interface to the registry using the wizard if it does not appear in the list of interfaces you can implement.

Once you create the subscriber object, you must subscribe to the event object's interface or to individual methods (events) on that interface. There are three types of subscriptions from which you can choose:

- **Transient subscriptions.** Like traditional event sinks, transient subscriptions are tied to the lifetime of an object instance. When the subscriber object is freed, the subscription ends and COM+ no longer forwards events to it.
- **Persistent subscriptions**. These are tied to the object class rather than a specific object instance. When the event occurs, COM locates or launches an instance of the subscriber object and calls its event handler. In-process objects (DLLs) use this type of subscription.
- **Per-user subscriptions**. These subscriptions provide a more secure version of transient subscriptions. Both the subscriber object and the server object that fires events must be running under the same user account on the same machine.

  Note: Objects that subscribe to COM+ events must be installed in a COM+ application.

See Also

Controlling an Automation Server Using a Dispatch Interface (see page 1424)

Controlling an Automation Server Using a Dual Interface (see page 1424)

3.2.2.4.11 Creating Clients for Servers That Do Not Have a Type Library

Some older COM technologies, such as object linking and embedding (OLE), do not provide type information in a type library. Instead, they rely on a standard set of predefined interfaces. To write clients that host such objects, you can use the TOleContainer component. This component appears on the System category of the **Tool Palette.**

TOleContainer acts as a host site for an Ole2 object. It implements the *IOleClientSite* interface and, optionally, *IOleDocumentSite*. Communication is handled using OLE verbs.

To use TOleContainer

1. Place a *TOleContainer* component on your form.
2. Set the *AllowActiveDoc* property to *true* if you want to host an Active document.
3. Set the *AllowInPlace* property to indicate whether the hosted object should appear in the *TOleContainer*, or in a separate window.
4. Write event handlers to respond when the object is activated, deactivated, moved, or resized.

5. To bind the *TOleContainer* object at design time, right click and choose Insert Object. In the Insert Object dialog, choose a server object to host.
6. To bind the *TOleContainer* object at runtime, you have several methods to choose from, depending on how you want to identify the server object. These include CreateObject, which takes a program id, CreateObjectFromFile, which takes the name of a file to which the object has been saved, CreateObjectFromInfo, which takes a record containing information on how to create the object, or CreateLinkToFile, which takes the name of a file to which the object was saved and links to it rather than embeds it.
7. Once the object is bound, you can access its interface using the OleObjectInterface property. However, because communication with Ole2 objects was based on OLE verbs, you will most likely want to send commands to the server using the DoVerb method.
8. When you want to release the server object, call the DestroyObject method.

See Also

Using Component Wrappers (see page 1422)

Writing Client Code Based On Type Library Definitions (see page 1423)

3.2.2.5 Creating simple COM servers

Topics

Name	Description
Creating Simple COM Servers: Overview (see page 1430)	Delphi provides wizards to help you create various COM objects. The simplest COM objects are servers that expose properties and methods (and possibly events) through a default interface that clients can call. Two wizards, in particular, ease the process of creating simple COM objects: • The COM Object wizard (see page 1431) builds a lightweight COM object whose default interface descends from IUnknown (see page 1387) or that implements an interface already registered on your system. This wizard provides the most flexibility in the types of COM objects you can create. • The Automation Object wizard (see page 1432) creates a simple Automation object whose default interface descends from IDispatch (see page 1438).... more (see page 1430)
Designing a COM Object (see page 1430)	When designing the COM object, you need to decide what COM interfaces you want to implement. You can write a COM object to implement an interface that has already been defined, or you can define a new interface for your object to implement. In addition, you can have your object support more than one interface. For information about standard COM interfaces that you might want to support, see the MSDN documentation. • To create a COM object that implements an existing interface, use the COM Object wizard (see page 1431). • To create a COM object that implements a new interface that you define,... more (see page 1430)

Using the COM Object Wizard (see page 1431)	The COM object wizard performs the following tasks: • Creates a new unit. • Defines a new class that descends from TCOMObject and sets up the class factory constructor. For more information on the base class, see Code generated by wizards (see page 1398). • Adds a type library to your project and adds your object and its interface to the type library. • Opens the type library in the Type Library Editor Before you create a COM object, create or open the project for the application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on... more (see page 1431)
Using the Automation Object Wizard (see page 1432)	The Automation object wizard performs the following tasks: • Creates a new unit. • Defines a new class that descends from TAutoObject and sets up the class factory constructor. For more information on the base class, see Code generated by wizards (see page 1398). • Adds a type library to your project and adds your object and its interface to the type library. Before you create an Automation object, create or open the project for an application containing functionality that you want to expose. The project can be either an application or ActiveX library, depending on your needs.
COM Object Instancing Types (see page 1433)	Many of the COM wizards require you to specify an instancing mode for the object. Instancing determines how many instances of your object clients can create in a single executable. If you specify a Single Instance model, for example, then once a client has instantiated your object, COM removes the application from view so that other clients must launch their own instances of the application. Because this affects the visibility of your application as a whole, the instancing mode must be consistent across all objects in your application that can be instantiated by clients. That is, you should not create... more (see page 1433)
Choosing a Threading Model (see page 1433)	When creating an object using a wizard, you select a threading model that your object agrees to support. By adding thread support to your COM object, you can improve its performance, because multiple clients can access your application at the same time. The following table lists the different threading models you can specify. ***Threading models for COM objects***
Defining a COM Object's Interface (see page 1435)	When you use a wizard to create a COM object, the wizard automatically generates a type library (unless you specify otherwise in the COM object wizard). The type library provides a way for host applications to find out what the object can do. It also lets you define your object's interface using the Type Library editor (see page 1446). The interfaces you define in the Type Library editor define what properties, methods, and events your object exposes to clients. **Note:** If you selected an existing interface in the COM object wizard, you do not need to add properties and methods. The definition... more (see page 1435)
Managing Events in Your Automation Object (see page 1437)	The Automation wizard automatically generates event code if you check the option, Generate Support Code in the Automation Object wizard dialog box. For a server to support traditional COM events, it must provide the definition of an outgoing interface which is implemented by a client. This outgoing interface includes all the event handlers the client must implement to respond to server events. When a client has implemented the outgoing event interface, it registers its interest in receiving event notification by querying the server's *IConnectionPointContainer* interface. The *IConnectionPointContainer* interface returns the server's *IConnectionPoint* interface, which the client then uses to pass... more (see page 1437)

Automation Interfaces (see page 1438)	The **Automation Object** wizard implements a dual interface (see page 1438) by default, which means that the Automation object supports both • Late binding at runtime, which is through the *IDispatch* interface. This is implemented as a dispatch interface, or dispinterface (see page 1438). • Early binding at compile-time, which is accomplished through directly calling one of the member functions in the object's virtual function table (VTable). This is referred to as a custom interface (see page 1439). **Note:** Any interfaces generated by the COM Object wizard that do not descend from *IDispatch* only support VTable calls.
Dual Interfaces (see page 1438)	A dual interface is a custom interface and a dispinterface at the same time. It is implemented as a COM VTable interface that derives from *IDispatch. For* those controllers that can access the object only at runtime, the dispinterface is available. For objects that can take advantage of compile-time binding, the more efficient VTable interface is used. Dual interfaces offer the following combined advantages of VTable interfaces and dispinterfaces: • For VTable interfaces, the compiler performs type checking and provides more informative error messages. • For Automation controllers that cannot obtain type information, the dispinterface provides runtime access to the object. • For... more (see page 1438)
Dispatch Interfaces (see page 1438)	*Automation controllers are clients that use the COM IDispatch interface to access the COM server objects. The controller must first create the object, then query the object's IUnknown interface for a pointer to its IDispatch interface. IDispatch keeps track of methods and properties internally by a dispatch identifier (dispID), which is a unique identification number for an interface member. Through IDispatch, a controller retrieves the object's type information for the dispatch interface and then maps interface member names to specific dispIDs. These dispIDs are available at runtime, and controllers get them by calling the IDispatch method,GetIDsOfNames.* *Once... more (see page 1438)*
Custom Interfaces (see page 1439)	Custom interfaces are user-defined interfaces that allow clients to invoke interface methods based on their order in the VTable and knowledge of the argument types. The VTable lists the addresses of all the properties and methods that are members of the object, including the member functions of the interfaces that it supports. If the object does not support *IDispatch*, the entries for the members of the object's custom interfaces immediately follow the members of *IUnknown*. If the object has a type library, you can access the custom interface through its VTable layout, which you can get using the... more (see page 1439)
Marshaling Data (see page 1439)	For out-of-process and remote servers, you must consider how COM marshals data outside the current process. You can provide marshaling: • Automatically, using the *IDispatch* interface. • Automatically, by creating a type library with your server and marking the interface with the OLE Automation flag. COM knows how to marshal all the **Automation-compatible** types in the type library and can set up the proxies and stubs for you. Some type restrictions apply to enable automatic marshaling. • Manually by implementing all the methods of the *IMarshal* interface. This is called **custom marshaling**. **Note:** The first method (using *IDispatch*) is only available... more (see page 1439)

Registering a COM Object (see page 1440)	You can register your server object as an in-process or an out-of-process server. For more information on the server types, see In-process (see page 1389). **Note:** Before you remove a COM object from your system, you should unregister it.
Testing and Debugging the Application (see page 1440)	Once you have created a COM server application, you will want to test it before you deploy it.

3.2.2.5.1 Creating Simple COM Servers: Overview

Delphi provides wizards to help you create various COM objects. The simplest COM objects are servers that expose properties and methods (and possibly events) through a default interface that clients can call.

Two wizards, in particular, ease the process of creating simple COM objects:

- The COM Object wizard (see page 1431) builds a lightweight COM object whose default interface descends from IUnknown (see page 1387) or that implements an interface already registered on your system. This wizard provides the most flexibility in the types of COM objects you can create.
- The Automation Object wizard (see page 1432) creates a simple Automation object whose default interface descends from IDispatch (see page 1438). *IDispatch* introduces a standard marshaling mechanism and support for late binding of interface calls.

  Note: COM defines many standard interfaces and mechanisms for handling specific situations. The Delphi wizards automate the most common tasks. However, some tasks, such as custom marshaling, are not supported by any Delphi wizards. For information on that and other technologies not explicitly supported by Delphi, refer to the Microsoft Developer's Network (MSDN) documentation. The Microsoft Web site also provides current information on COM support.

Overview of creating a COM object

Whether you use the **Automation Object wizard** to create a new Automation server or the COM object wizard to create some other type of COM object, the process you follow is the same.

It involves these steps:

1. Design (see page 1430) the COM object.
2. Use the COM Object wizard (see page 1431) or the Automation Object wizard (see page 1432) to create the server object.
3. Define the interface (see page 1435) that the object exposes to clients.
4. Register the COM object (see page 1440).
5. Test and debug (see page 1440) the application.

See Also

Overview of COM Technologies (see page 1385)

Creating Active Server Pages (see page 1400)

3.2.2.5.2 Designing a COM Object

When designing the COM object, you need to decide what COM interfaces you want to implement. You can write a COM object to implement an interface that has already been defined, or you can define a new interface for your object to implement. In addition, you can have your object support more than one interface. For information about standard COM interfaces that you might want to support, see the MSDN documentation.

- To create a COM object that implements an existing interface, use the COM Object wizard (see page 1431).
- To create a COM object that implements a new interface that you define, use either the COM Object wizard (see page 1431) or the Automation Object wizard (see page 1432). The COM object wizard can generate a new default interface that descends from IUnknown (see page 1387), and the Automation object gives your object a default interface that descends from IDispatch (see page 1438). No matter which wizard you use, you can always use the Type Library editor later to

change the parent interface of the default interface that the wizard generates.

In addition to deciding what interfaces to support, you must decide whether the COM object is an in-process server (see page 1389). For in-process servers and for out-of-process and remote servers that use a type library, COM marshals the data for you. Otherwise, you must consider how to marshal the data to out-of-process servers.

3.2.2.5.3 Using the COM Object Wizard

The COM object wizard performs the following tasks:

- Creates a new unit.
- Defines a new class that descends from TCOMObject and sets up the class factory constructor. For more information on the base class, see Code generated by wizards (see page 1398).
- Adds a type library to your project and adds your object and its interface to the type library.
- Opens the type library in the Type Library Editor

Before you create a COM object, create or open the project for the application containing functionality that you want to implement. The project can be either an application or ActiveX library, depending on your needs.

To bring up the COM object wizard

1. Choose **File** ▶ **New** ▶ **Other** to open the New Items dialog box.
2. Select the folder labeled **ActiveX** under **Delphi Projects**
3. Double-click the COM object icon in the right pane.

In the wizard, you must specify the following:

- **CoClass name:** This is the name of the object as it appears to clients. The class created to implement your object has this name with a 'T' prepended. If you do not choose to implement an existing interface, the wizard gives your CoClass a default interface that has this name with an 'I' prepended.
- Instancing (see page 1433): Unless you are creating an in-process server, you need to indicate how COM launches the application that houses your COM object. If your application implements more than one COM object, you should specify the same instancing for all of them.
- Threading Model (see page 1433): Typically, client requests to your object enter on different threads of execution. You can specify how COM serializes these threads when it calls your object. Your choice of threading model determines how the object is registered. You are responsible for providing any threading support implied by the model you choose. For information on how to provide thread support to your application, see Writing multi-threaded applications
- **Interface:** The wizard gives your object a default interface that descends from IUnknown (see page 1387). By default, the wizard gives your interface the same name as the CoClass, preceded by “I” to indicate interface. After exiting the wizard, you can then use the Type Library editor to add properties and methods to this interface. However, you can also select a pre-defined interface for your object to implement. Click the **[...]** button in the COM object wizard to bring up the Interface Selection wizard, where you can select any dual or custom interface (see page 1438) defined in a type library registered on your system. The interface you select becomes the default interface for your new CoClass. The wizard adds all the methods on this interface to the generated implementation class, so that you only need to fill in the bodies of the methods in the implementation unit. Note that if you select an existing interface, the interface is not added to your project's type library. This means that when deploying your object, you must also deploy the type library that defines the interface.
- Include Type Library (see page 1395): You can choose whether you want to include a type library for your object. This is recommended for two reasons: it lets you use the Type Library editor to define interfaces, thereby updating much of the implementation, and it gives clients an easy way to obtain information about your object and its interfaces. If you are implementing an existing interface, Delphi requires your project to use a type library. This is the only way to provide access to the original interface declaration.
- Mark interface Oleautomation (see page 1439): If you have opted to create a type library and are willing to confine yourself to Automation-compatible types, you can let COM handle the marshaling for you when you are not generating an in-process server. By marking your object's interface as OleAutomation in the type library, you enable COM to set up the proxies and stubs for you and handles passing parameters across process boundaries. You can only specify whether your interface is Automation-compatible if you are generating a new interface. If you select an existing interface, its attributes are already

specified in its type library. If your object's interface is not marked as OleAutomation, you must either create an in-process server or write your own marshaling code.

You can optionally add a description of your COM object. This description appears in the type library for your object if you create one.

See Also

Implementing COM objects with Wizards (see page 1397)

Designing a COM Object (see page 1430)

Defining a COM Object's Interface (see page 1435)

Registering a COM Object (see page 1440)

3.2.2.5.4 Using the Automation Object Wizard

The Automation object wizard performs the following tasks:

- Creates a new unit.
- Defines a new class that descends from TAutoObject and sets up the class factory constructor. For more information on the base class, see Code generated by wizards (see page 1398).
- Adds a type library to your project and adds your object and its interface to the type library.

Before you create an Automation object, create or open the project for an application containing functionality that you want to expose. The project can be either an application or ActiveX library, depending on your needs.

To display the Automation wizard:

1. Choose **File** ▶ **New** ▶ **Other** to open the New Items dialog box.
2. Select the folder labeled **ActiveX** under **Delphi Projects**.
3. Double-click the **Automation Object** icon in the right pane.
4. In the wizard dialog, specify the following:

- **CoClass name:** This is the name of the object as it appears to clients. Your object's default interface is created with a name based on this CoClass name with an 'I' prepended, and the class created to implement your object has this name with a 'T' prepended.
- Instancing (see page 1433): Unless you are creating an in-process server, you need to indicate how COM launches the application that houses your COM object. If your application implements more than one COM object, you should specify the same instancing for all of them.
- Threading Model (see page 1433): Typically, client requests to your object enter on different threads of execution. You can specify how COM serializes these threads when it calls your object. Your choice of threading model determines how the object is registered. You are responsible for providing any threading support implied by the model you choose. For information on how to provide thread support to your application, see Writing multi-threaded applications.
- Generate Event support code (see page 1437): You must indicate whether you want your object to generate events to which clients can respond. The wizard can provide support for the interfaces required to generate events and the dispatching of calls to client event handlers.

The Automation object implements a dual interface (see page 1438), which supports both early (compile-time) binding through the VTable and late (runtime) binding through the IDispatch interface.

See Also

Implementing COM objects with Wizards (see page 1397)

Designing a COM Object (see page 1430)

Defining a COM Object's Interface (see page 1435)

Registering a COM Object (see page 1440)

Using the COM Object Wizard (see page 1431)

3.2.2.5.5 COM Object Instancing Types

Many of the COM wizards require you to specify an instancing mode for the object. Instancing determines how many instances of your object clients can create in a single executable. If you specify a Single Instance model, for example, then once a client has instantiated your object, COM removes the application from view so that other clients must launch their own instances of the application. Because this affects the visibility of your application as a whole, the instancing mode must be consistent across all objects in your application that can be instantiated by clients. That is, you should not create one object in your application that uses Single Instance mode and another in the same application that uses Multiple Instance mode.

Note: Instancing is ignored when your COM object is used only as an in-process server.

When the wizard creates a new COM object, it can have any of the following instancing types:

Instancing	Meaning
Internal	The object can only be created internally. An external application cannot create an instance of the object directly, although your application can create the object and pass an interface for it to clients.
Single Instance	Allows clients to create only a single instance of the object for each executable (application), so creating multiple instances results in launching multiple instances of the application. Each client has its own dedicated instance of the server application.
Multiple Instances	Specifies that multiple clients can create instances of the object in the same process space.

3.2.2.5.6 Choosing a Threading Model

When creating an object using a wizard, you select a threading model that your object agrees to support. By adding thread support to your COM object, you can improve its performance, because multiple clients can access your application at the same time.

The following table lists the different threading models you can specify.

Threading models for COM objects

Threading model	Description	Implementation pros and cons
Single	The server provides no thread support. COM serializes client requests so that the application receives one request at a time.	Clients are handled one at a time so no threading support is needed. No performance benefit.
Apartment (or Single-threaded apartment)	COM ensures that only one client thread can call the object at a time. All client calls use the thread in which the object was created.	Objects can safely access their own instance data, but global data must be protected using critical sections or some other form of serialization. The thread's local variables are reliable across multiple calls. Some performance benefits.
Free (also called multi-threaded apartment)	Objects can receive calls on any number of threads at any time.	Objects must protect all instance and global data using critical sections or some other form of serialization. Thread local variables are not reliable across multiple calls.

Both	This is the same as the Free-threaded model except that outgoing calls (for example, callbacks) are guaranteed to execute in the same thread.	Maximum performance and flexibility. Does not require the application to provide thread support for parameters supplied to outgoing calls.
Neutral	Multiple clients can call the object on different threads at the same time, but COM ensures that no two calls conflict.	You must guard against thread conflicts involving global data and any instance data that is accessed by multiple methods. This model should not be used with objects that have a user interface (visual controls). This model is only available under COM+. Under COM, it is mapped to the Apartment model.

Note: Local variables (except those in callbacks) are always safe, regardless of the threading model. This is because local variables are stored on the stack and each thread has its own stack. Local variables may not be safe in callbacks when using free-threading.

The threading model you choose in the wizard determines how the object is registered in the system Registry. You must make sure that your object implementation adheres to the threading model you have chosen. For general information on writing thread-safe code, see Writing multi-threaded applications.

For in-process servers, setting the threading model in the wizard sets the threading model key in the CLSID registry entry.

Out-of-process servers are registered as EXE, and Delphi initializes COM for the highest threading model required. For example, if an EXE includes a free-threaded object, it is initialized for free threading, which means that it can provide the expected support for any free-threaded or apartment-threaded objects contained in the EXE. To manually override threading behavior in EXEs, use the CoInitFlags variable.

Writing an object that supports the free threading model

Use the free threading (or both) model rather than apartment threading whenever the object needs to be accessed from more than one thread. A common example is a client application connected to an object on a remote machine. When the remote client calls a method on that object, the server receives the call on a thread from the thread pool on the server machine. This receiving thread makes the call locally to the actual object; and, because the object supports the free threading model, the thread can make a direct call into the object.

If the object supported the apartment threading model instead, the call would have to be transferred to the thread on which the object was created, and the result would have to be transferred back into the receiving thread before returning to the client. This approach requires extra marshaling.

To support free threading, you must consider how instance data can be accessed for *each* method. If the method is writing to instance data, you must use critical sections or some other form of serialization, to protect the instance data. Likely, the overhead of serializing critical calls is less than executing COM's marshaling code.

Note that if the instance data is read-only, serialization is not needed.

Free-threaded in-process servers can improve performance by acting as the outer object in an aggregation (see page 1391) with the free-threaded marshaler. The free-threaded marshaler provides a shortcut for COM's standard thread handling when a free-threaded DLL is called by a host (client) that is not free-threaded.

To aggregate with the free threaded marshaler, you must

- Call *CoCreateFreeThreadedMarshaler*, passing your object's *IUnknown* interface for the resulting free-threaded marshaler to use: `CoCreateFreeThreadedMarshaler(self as IUnknown, FMarshaler);` `CoCreateFreeThreadedMarshaler(static_cast<IUnknown *>(this), &FMarshaler);`. This line assigns the interface for the free-threaded marshaler to a class member, *FMarshaler*.
- Using the **Type Library Editor**, add the *IMarshal* interface to the set of interfaces your CoClass implements.
- In your object's *QueryInterface* method, delegate calls for IDD_IMarshal to the free-threaded marshaler (stored as *FMarshaler* above).

Warning: The free-threaded marshaler violates the normal rules of COM marshaling to provide additional efficiency. It should be used with care. In particular, it should only be aggregated with free-threaded objects in an in-process server, and should only be instantiated by the object that uses it (not another thread).

Writing an object that supports the apartment threading model

To implement the (single-threaded) apartment threading model, you must follow a few rules:

- The first thread in the application that gets created is COM's main thread. This is typically the thread on which WinMain was called. This must also be the last thread to uninitialize COM.
- Each thread in the apartment threading model must have a message loop, and the message queue must be checked frequently.
- When a thread gets a pointer to a COM interface, that pointer may only be used in that thread.

The single-threaded apartment model is the middle ground between providing no threading support and full, multi-threading support of the free threading model. A server committing to the apartment model promises that the server has serialized access to all of its global data (such as its object count). This is because different objects may try to access the global data from different threads. However, the object's instance data is safe because the methods are always called on the same thread.

Typically, controls for use in Web browsers use the apartment threading model because browser applications always initialize their threads as apartment.

Writing an object that supports the neutral threading model

Under COM+, you can use another threading model that is in between free threading and apartment threading: the neutral model. Like the free-threading model, this model allows multiple threads to access your object at the same time. There is no extra marshaling to transfer to the thread on which the object was created. However, your object is guaranteed to receive no conflicting calls.

Writing an object that uses the neutral threading model follows much the same rules as writing an apartment-threaded object, except that you do need to guard instance data against thread conflicts if it can be accessed by different methods in the object's interface. Any instance data that is only accessed by a single interface method is automatically thread-safe.

3.2.2.5.7 Defining a COM Object's Interface

When you use a wizard to create a COM object, the wizard automatically generates a type library (unless you specify otherwise in the COM object wizard). The type library provides a way for host applications to find out what the object can do. It also lets you define your object's interface using the Type Library editor (see page 1446). The interfaces you define in the Type Library editor define what properties, methods, and events your object exposes to clients.

Note: If you selected an existing interface in the COM object wizard, you do not need to add properties and methods. The definition of the interface is imported from the type library in which it was defined. Instead, simply locate the methods of the imported interface in the implementation unit and fill in their bodies.

Adding a property to the object's interface

When you add a property (see page 1462) to your object's interface using the **Type Library Editor**, it automatically adds a method to read the property's value and/or a method to set the property's value. The **Type Library Editor**, in turn, adds these methods to your implementation class, and in your implementation unit creates empty method implementations for you to complete.

To add a property to your object's interface

1. In the **Type Library Editor**, select the default interface for the object. The default interface should be the name of the object preceded by the letter "I." To determine the default, in the **Type Library Editor**, click the CoClass and then select the **Implements** tab, and check the list of implemented interfaces for the one marked, "Default."
2. To expose a read/write property, click the **New Property** button on the toolbar; otherwise, click the arrow next to the **New Property** button on the toolbar, and then click the type of property to expose.

3. In the **Attributes** pane, specify the name and type of the property.
4. On the **Type Library Editor** toolbar, click the **Refresh Implementation** button. A definition and skeletal implementations for the property access methods are inserted into the object's implementation unit.
5. In the implementation unit, locate the access methods for the property. These have names of the form `Get_PropertyName` and `Set_PropertyName`. Add code that gets or sets the property value of your object. This code may simply call an existing function inside the application, access a data member that you add to the object definition, or otherwise implement the property.

Adding a method to the object's interface

When you add a method (see page 1462) to your object's interface using the **Type Library Editor**, the **Type Library Editor** can, in turn, add the methods to your implementation class, and in your implementation unit create empty implementation for you to complete.

To expose a method via your object's interface

1. In the **Type Library Editor**, select the default interface for the object. The default interface should be the name of the object preceded by the letter "I". To determine the default, in the **Type Library Editor**, click the CoClass and select the **Implements** tab, and check the list of implemented interfaces for the one marked, "Default."
2. Click the **New Method** button.
3. In the **Attributes** pane, specify the name of the method.
4. In the **Parameters** pane, specify the method's return type and add the appropriate parameters.
5. On the **Type Library Editor** toolbar, click the **Refresh Implementation** button. A definition and skeletal implementation for the method is inserted into the object's implementation unit.
6. In the implementation unit, locate the newly inserted method implementation. The method is completely empty. Fill in the body to perform whatever task the method represents.

Exposing events to clients

There are two types of events that a COM object can generate: traditional events and COM+ events.

- COM+ events require that you create a separate event object using the event object wizard and add code to call that event object from your server object.
- You can use the wizard to handle much of the work in generating traditional events. This process is described below.

  Note: The COM object wizard does not generate event support code. If you want your object to generate traditional events, you should use the Automation object wizard

  .

In order for an object to generate events, you need to do the following:

1. In the **Automation Object wizard**, check the box, **Generate event support code**. The wizard creates an object that includes an Events interface as well as the default interface. This Events interface has a name of the form `I`*`CoClassname`*`Events`. It is an outgoing (source) interface, which means that it is not an interface your object implements, but rather is an interface that clients must implement and which your object calls. (You can see this by selecting your CoClass, going to the **Implements** page, and noting that the **Source** column on the Events interface says true.) In addition to the Events interface, the wizard adds the *IConnectionPointContainer* interface to the declaration of your implementation class, and adds several class members for handling events. Of these new class members, the most important are *FConnectionPoint* and *FConnectionPoints*, which implement the *IConnectionPoint* and *IConnectionPointContainer* interfaces using built-in VCL classes. *FConnectionPoint* is maintained by another method that the wizard adds, *EventSinkChanged*.
2. In the **Type Library Editor**, select the outgoing Events interface for your object. (This is the one with a name of the form `I`*`CoClassName`*`Events`)
3. Click the **New Method** button from the **Type Library Editor** toolbar. Each method you add to the Events interface represents an event handler that the client must implement.
4. In the **Attributes** pane, specify the name of the event handler, such as `MyEvent`.

5. On the **Type Library Editor** toolbar, click the **Refresh Implementation** button. Your object implementation now has everything it needs to accept client event sinks and maintain a list of interfaces to call when the event occurs. To call these interfaces, you can create a method to generate each event on clients.
6. In the **Code Editor**, add a method to your object for firing each event. For example,

```
unit ev;
interface
uses
   ComObj, AxCtrls, ActiveX, Project1_TLB;
type
   TMyAutoObject = class (TAutoObject,IConnectionPointContainer, IMyAutoObject)
private
   .
   .
   .
public
   procedure Initialize; override;
   procedure Fire_MyEvent; { Add a method to fire the event}
```

7. Implement the method you added in the last step so that it iterates through all the event sinks maintained by your object's *FConnectionPoint* member:

```
procedure TMyAutoObject.Fire_MyEvent;
var
  I: Integer;
  EventSinkList: TList;
  EventSink: IMyAutoObjectEvents;
begin
  if FConnectionPoint <> nil then
  begin
    EventSinkList :=FConnectionPoint.SinkList; {get the list of client sinks }
    for I := 0 to EventSinkList.Count - 1 do
    begin
      EventSink := IUnknown(FEvents[I]) as IMyAutoObjectEvents;
      EventSink.MyEvent;
    end;
  end;
end;
```

8. Whenever you need to fire the event so that clients are informed of its occurrence, call the method that dispatches the event to all event sinks:

```
if EventOccurs then Fire_MyEvent; { Call method you created to fire events.}
if (EventOccurs) Fire_MyEvent; // Call method you created to fire events.
```

See Also

Working with Type Libraries: Overview (see page 1445)

3.2.2.5.8 Managing Events in Your Automation Object

The Automation wizard automatically generates event code if you check the option, Generate Support Code in the Automation Object wizard dialog box.

For a server to support traditional COM events, it must provide the definition of an outgoing interface which is implemented by a client. This outgoing interface includes all the event handlers the client must implement to respond to server events.

When a client has implemented the outgoing event interface, it registers its interest in receiving event notification by querying the server's *IConnectionPointContainer* interface. The *IConnectionPointContainer* interface returns the server's *IConnectionPoint* interface, which the client then uses to pass the server a pointer to its implementation of the event handlers (known as a sink).

The server maintains a list of all client sinks and calls methods on them when an event occurs.

When you select Generate Event Support Code, Delphi automatically generates the code necessary to support IConnectionPoint

and IConnectionPointContainer. This support, and the way you can use it to generate events is described in Exposing events to clients (see page 1435).

See Also

Using the Automation Object Wizard (see page 1432)

3.2.2.5.9 Automation Interfaces

The **Automation Object** wizard implements a dual interface (see page 1438) by default, which means that the Automation object supports both

- Late binding at runtime, which is through the *IDispatch* interface. This is implemented as a dispatch interface, or dispinterface (see page 1438).
- Early binding at compile-time, which is accomplished through directly calling one of the member functions in the object's virtual function table (VTable). This is referred to as a custom interface (see page 1439).

  Note: Any interfaces generated by the COM Object

  wizard that do not descend from *IDispatch* only support VTable calls.

3.2.2.5.10 Dual Interfaces

A dual interface is a custom interface and a dispinterface at the same time. It is implemented as a COM VTable interface that derives from *IDispatch. For* those controllers that can access the object only at runtime, the dispinterface is available. For objects that can take advantage of compile-time binding, the more efficient VTable interface is used.

Dual interfaces offer the following combined advantages of VTable interfaces and dispinterfaces:

- For VTable interfaces, the compiler performs type checking and provides more informative error messages.
- For Automation controllers that cannot obtain type information, the dispinterface provides runtime access to the object.
- For in-process servers, you have the benefit of fast access through VTable interfaces.
- For out-of-process servers, COM marshals data (see page 1439) for both VTable interfaces and dispinterfaces. COM provides a generic proxy/stub implementation that can marshal the interface based on the information contained in a type library.

The first three entries of the VTable for a dual interface refer to the *IUnknown* interface, the next four entries refer to the *IDispatch* interface, and the remaining entries are COM entries for direct access to members of the custom interface.

3.2.2.5.11 Dispatch Interfaces

Automation controllers are clients that use the COM IDispatch interface to access the COM server objects. The controller must first create the object, then query the object's IUnknown interface for a pointer to its IDispatch interface. IDispatch keeps track of methods and properties internally by a dispatch identifier (dispID), which is a unique identification number for an interface member. Through IDispatch, a controller retrieves the object's type information for the dispatch interface and then maps interface member names to specific dispIDs. These dispIDs are available at runtime, and controllers get them by calling the IDispatch method,GetIDsOfNames.

Once it has the dispID, the controller can then call the IDispatch method, Invoke, to execute the appropriate code (property or method), packaging the parameters for the property or method into one of the Invoke parameters. Invoke has a fixed compile-time signature that allows it to accept any number of arguments when calling an interface method.

The Automation object's implementation of Invoke must then unpackage the parameters, call the property or method, and be prepared to handle any errors that occur. When the property or method returns, the object passes its return value back to the controller.

This is called late binding because the controller binds to the property or method at runtime rather than at compile time.

3.2.2.5.12 Custom Interfaces

Custom interfaces are user-defined interfaces that allow clients to invoke interface methods based on their order in the VTable and knowledge of the argument types. The VTable lists the addresses of all the properties and methods that are members of the object, including the member functions of the interfaces that it supports. If the object does not support *IDispatch*, the entries for the members of the object's custom interfaces immediately follow the members of *IUnknown*.

If the object has a type library, you can access the custom interface through its VTable layout, which you can get using the **Type Library Editor**. If the object has a type library and also supports *IDispatch*, a client can also get the dispIDs of the *IDispatch* interface and bind directly to a VTable offset. Delphi's type library importer (TLIBIMP) retrieves dispIDs at import time, so clients that use dispinterfaces can avoid calls to *GetIDsOfNames; this* information is already in the _TLB unit. However, clients still need to call *Invoke*.

3.2.2.5.13 Marshaling Data

For out-of-process and remote servers, you must consider how COM marshals data outside the current process. You can provide marshaling:

- Automatically, using the *IDispatch* interface.
- Automatically, by creating a type library with your server and marking the interface with the OLE Automation flag. COM knows how to marshal all the **Automation-compatible** types in the type library and can set up the proxies and stubs for you. Some type restrictions apply to enable automatic marshaling.
- Manually by implementing all the methods of the *IMarshal* interface. This is called **custom marshaling**.

  Note: The first method (using *IDispatch*) is only available on Automation servers. The second method is automatically available on all objects that are created by wizards and which use a type library.

Automation compatible types

Function result and parameter types of the methods declared in dual and dispatch interfaces and interfaces that you mark as OLE Automation must be *Automation-compatible* types. The following types are OLE Automation-compatible:

First, the predefined valid types such as *Smallint*, *Integer*, *Single*, *Double*, *WideString*. For a complete list, see Valid types (see page 1454).

Second, enumeration types defined in a type library. OLE Automation-compatible enumeration types are stored as 32-bit values and are treated as values of type *Integer* for purposes of parameter passing.

Third, interface types defined in a type library that are OLE Automation safe, that is, derived from *IDispatch* and containing only OLE Automation compatible types.

Fourth, dispinterface types defined in a type library.

Fifth, any custom record type defined within the type library.

Sixth, *IFont*, *IStrings*, and *IPicture*. Helper objects must be instantiated to map

- an *IFont to* a *TFont*
- an *IStrings* to a *TStrings*
- an *IPicture* to a *TPicture*

The ActiveX control and ActiveForm wizards create these helper objects automatically when needed. To use the helper objects, call the global routines, GetOleFont, GetOleStrings, GetOlePicture, respectively.

Type restrictions for automatic marshaling

For an interface to support automatic marshaling (also called Automation marshaling or type library marshaling), the following restrictions apply. When you edit your object using the type library editor, the editor enforces these restrictions:

- String data types must be transferred as wide strings (BSTR). **PChar**, **UnicodeString** and **AnsiString** cannot be marshaled safely.
- All members of a dual interface must pass an HRESULT as the function's return value. If the method is declared using the safecall calling convention, this condition is imposed automatically, with the declared return type converted to an output parameter.
- Members of a dual interface that need to return other values should specify these parameters as **var** or **out**, indicating an output parameter that returns the value of the function.

  Note: One way to bypass the Automation types restrictions is to implement a separate *IDispatch* interface and a custom interface. By doing so, you can use the full range of possible argument types. This means that COM clients have the option of using the custom interface, which Automation controllers can still access. In this case, though, you must implement the marshaling code manually.

Custom marshaling

Typically, you use automatic marshaling in out-of-process and remote servers because it is easier—COM does the work for you. However, you may decide to provide custom marshaling if you think you can improve marshaling performance. When implementing your own custom marshaling, you must support the *IMarshal* interface. For more information, on this approach, see the Microsoft documentation.

See Also

Working with Type Libraries: Overview (see page 1445)

Automation Interfaces (see page 1438)

3.2.2.5.14 Registering a COM Object

You can register your server object as an in-process or an out-of-process server. For more information on the server types, see In-process (see page 1389).

Note: Before you remove a COM object from your system, you should unregister it.

Registering an in-process server

To register an in-process server (DLL or OCX), choose **Run** ▶ **Register ActiveX Server**.

To unregister an in-process server, choose **Run** ▶ **Unregister ActiveX Server**.

Registering an out-of-process server

To register an out-of-process server, run the server with the **/regserver** command-line option. You can set command-line options with the **Run** ▶ **Parameters** dialog box. You can also register the server by running it.

To unregister an out-of-process server, run the server with the **/unregserver** command-line option.

As an alternative, you can use the `tregsvr` command from the command line or run the `regsvr32` command from the operating system.

Note: If the COM server is intended for use under COM+, you should install it in a COM+ application rather than register it. (Installing the object in a COM+ application automatically takes care of registration.)

3.2.2.5.15 Testing and Debugging the Application

Once you have created a COM server application, you will want to test it before you deploy it.

To test and debug your COM server application,

1. Turn on debugging information using the **Compiler** page on the **Project ▸ Options** dialog box, if necessary. Also, turn on **Integrated Debugging** in the **Tools ▸ Options ▸ Debugger Options** dialog.
2. For an in-process server, choose **Run ▸ Parameters**, type the name of the Automation controller in the **Host Application** box, and choose OK.
3. Choose **Run ▸ Run**.
4. Set breakpoints in the Automation server.
5. Use the Automation controller to interact with the Automation server.

The Automation server pauses when the breakpoints are reached.

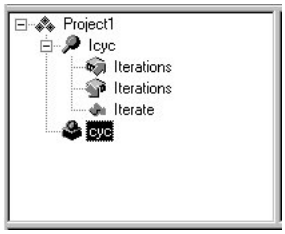
Note: As an alternate approach, if you are also writing the Automation controller, you can debug into an in-process server by enabling COM cross-process support. Use the CodeGear Debuggers

page of the **Tools ▸ Options ▸ Debugger Options** dialog to enable cross-process support.

3.2.2.6 Working with type libraries

Topics

Name	Description
Working with Type Libraries: Overview (see page 1445)	Type libraries are files that include information about data types, interfaces, member functions, and object classes exposed by a COM object. They provide a way to identify the types of objects and interfaces that are available on a server. For a detailed overview on why and when to use type libraries, see Type libraries (see page 1395). A type library can contain any and all of the following: • Information about custom data types such as aliases, enumerations, structures, and unions. • Descriptions of one or more COM elements, such as an interface, dispinterface, or CoClass. Each of these descriptions is commonly referred to... more (see page 1445)
Type Library Editor (see page 1446)	The **Type Library Editor** enables developers to examine and create type information for COM objects. Using the **Type Library Editor** can greatly simplify the task of developing COM objects by centralizing the tasks of defining interfaces, CoClasses, and types, obtaining GUIDs for new interfaces, associating interfaces with CoClasses, updating implementation units, and so on. The **Type Library Editor** outputs two types of file that represent the contents of the type library: ***Type Library editor files***
Parts of the Type Library Editor (see page 1446)	The main elements of the **Type Library Editor** are described in the following table: ***Type Library editor parts***
Toolbar (see page 1447)	The **Type Library Editor**'s toolbar, located at the top of the Type Library Editor, contains buttons that you click to add new objects into your type library. The first group of buttons let you add elements to the type library. When you click a toolbar button, the icon for that element appears in the object list pane. You can then customize its attributes in the right pane. Depending on the type of icon you select, different pages of information appear to the right. The following table lists the elements (see page 1451) you can add to your type library: ***Icons Representing... more (see page 1447)***

Object List Pane (see page 1448)	The **Object list** pane displays all the elements of the current type library in a tree view. The root of the tree represents the type library itself, and appears as the following icon: Descending from the type library node are the elements (see page 1451) in the type library:  When you select any of these elements (including the type library itself), the pages of type information (see page 1449) to the right change to reflect only the relevant information for that element. You can use these pages to edit the definition and properties of the selected element. You can manipulate the elements in the object... more (see page 1448)
Status Bar (see page 1449)	When editing or saving a type library, syntax, translation errors, and warnings are listed in the **Message** pane. For example, if you specify a type that the **Type Library Editor** does not support, you will get a syntax error. For a complete list of types supported by the **Type Library Editor**, see Valid types (see page 1454).
Pages of Type Information (see page 1449)	When you select an element (see page 1451) in the object list pane, pages of type information appear in the **Type Library Editor** that are valid for the selected element. Which pages appear depends on the element selected in the object list panel, as follows:
Type Library Elements (see page 1451)	The Type Library interface as represented in the Type Library Editor can seem overwhelmingly complicated at first. This is because it represents information about a great number of elements, each of which has its own characteristics. However, many of these characteristics are common to all elements. For example, every element (including the type library itself) has the following: • **Name**, which is used to describe the element and which is used when referring to the element in code. • **GUID** (globally unique identifier), which is a unique 128-bit value that COM uses to identify the element. This should always be supplied... more (see page 1451)
Using the Type Library Editor (see page 1453)	Using the type library editor, you can create new type libraries or edit existing ones. Typically, an application developer uses a wizard to create the objects that are exposed in the type library, letting Delphi generate the type library automatically. Then, the automatically-generated type library is opened in the Type Library editor so that the interfaces can be defined (or modified), type definitions added, and so on. However, even if you are not using a wizard to define the objects, you can use the Type Library editor to define a new type library. In this case, you must create any... more (see page 1453)
Valid Types (see page 1454)	In the Type Library editor, you use different type identifiers, depending on whether you are working in IDL or Delphi. Specify the language you want to use in the Environment options dialog. The following types are valid in a type library for COM development. The Automation compatible column specifies whether the type can be used by an interface that has its Automation or Dispinterface flag checked. These are the types that COM can marshal via the type library automatically.
SafeArrays (see page 1455)	COM requires that arrays be passed via a special data type known as a SafeArray. You can create and destroy SafeArrays by calling special COM functions to do so, and all elements within a SafeArray must be valid automation-compatible types (see page 1454). The Delphi compiler has built-in knowledge of COM SafeArrays and automatically calls the COM API to create, copy, and destroy SafeArrays. In the **Type Library Editor**, a *SafeArray* must specify the type of its elements. For example, the following line from the text page declares a method with a parameter that is a *SafeArray* with an element type... more (see page 1455)

Using Object Pascal or IDL Syntax (see page 1455)	The Code page of the Type Library editor displays your type information in RIDL format (Restricted Interface Definition Language). The format resembles Microsoft's IDL sytnax. However, RIDL supports only a subset of IDL. Like Delphi applications in general, identifiers in type libraries are case insensitive. Identifiers can be as many as 255 characters long, and for maximum portability should begin with a letter or an underscore (_).
Creating a New Type Library (see page 1460)	Use the **Type Library Wizard** to create a type library that is independent of a particular COM object. For example, you might want to define a type library that contains type definitions that you use in several other type libraries. You can then create a type library of basic definitions and add it to the `uses` page of other type libraries. You can also use the wizard to create a type library for an object that is not yet implemented. Once the type library contains the interface definition, you can use the COM object wizard (see page 1431) to generate a CoClass and... more (see page 1460)
Opening an Existing Type Library (see page 1460)	When you use the wizards to create an Automation object, COM object, transactional object, or a remote data module, a type library is automatically created with an implementation unit. Starting with the 2009 product, the wizards create type libraries in a text-based format (RIDL files). However, you can still open type libraries in `.tlb` format that are associated with previous versions or other products (servers) that are available on your system. The Type Library Explorer enables you to examine `.tlb` files.
Adding an Interface to the Type Library (see page 1461)	
Modifying an Interface Using the Type Library (see page 1461)	There are several ways to modify an interface or dispinterface once it is created. • You can change the interface's attributes using the page of type information that contains the information you want to change. Select the interface in the **Object List** pane and then use the controls on the appropriate page of type information. For example, you may want to change the parent interface using the attributes page, or use the flags page to change whether or not it is a dual interface. • You can edit the interface declaration directly by selecting the interface in the object list pane and... more (see page 1461)
Adding Properties and Methods to the Type Library (see page 1462)	
Adding a CoClass to the Type Library (see page 1463)	The easiest way to add a CoClass to your project is to choose **File** ▶ **New** ▶ **Other** from the main menu in the IDE and use the appropriate wizard (see page 1397) on the ActiveX page of the **New Items** dialog. The advantage to this approach is that, in addition to adding the CoClass and its interface to the type library, the wizard adds an implementation unit and updates the project file to include the new implementation unit in its uses clause. If you are not using a wizard, however, you can create a CoClass by clicking the CoClass icon on the toolbar and... more (see page 1463)
Adding an Interface to a CoClass (see page 1463)	CoClasses are defined by the interfaces they present to clients. While you can add any number of properties and methods to the implementation class of a CoClass, clients can only see those properties and methods that are exposed by interfaces associated with the CoClass. To associate an interface with a CoClass, right-click in the **Implements** page for the class and choose Insert Interface to display a list of interfaces from which you can choose. The list includes interfaces that are defined in the current type library and those defined in any type libraries that the current type library references. Choose... more (see page 1463)
Adding an Enumeration to the Type Library (see page 1464)	
Adding an Alias to the Type Library (see page 1464)	
Adding a Record or Union to the Type Library (see page 1464)	
Adding a Module to the Type Library (see page 1465)	

Saving and Registering Type Library Information (see page 1465)	After modifying your type library, you'll want to save and register the type library information. Saving the type library automatically updates: • Both the text-based type library file (`.ridl` extension) and the binary type library file (`.tlb` extension). • The *Project*_TLB unit that represents its contents • The implementation code for any CoClasses that were generated by a wizard. **Note:** The type library is stored as separate text (.RIDL) and binary (.TLB) files, but is also linked into the server (.EXE, DLL, or .OCX). The **Type Library Editor** gives you toolbar buttons for storing your type library information: • Refresh (see page 1467) updates... more (see page 1465)
Apply Updates Dialog (see page 1466)	The **Apply Updates** dialog appears when you refresh, register, or save the type library if you have selected **Display updates before refreshing** in the **Tools ▶ Options ▶ Type Library** page (which is not checked off by default). Without this option, the **Type Library Editor** automatically updates the sources of the associated object when you make changes in the editor. With this option, you have a chance to veto the proposed changes when you attempt to refresh, save, or register the type library. The **Apply Updates** dialog will warn you about potential errors, and will insert TODO comments in your source... more (see page 1466)
Saving a Type Library (see page 1466)	For 2009, RAD Studio stores type libraries as separate text-based files (RIDL, or Restricted Interface Definition Language); The RIDL file, which represents the TLB file in the Editor, is also linked into the server (.EXE, DLL, or .OCX). When you build your project, the RIDL file is saved on disk as both a `.tlb` file as well as a `.ridl` file. There are three ways to save a type library. • **File>Save Command:** To save the project and the type library, choose **File ▶ Save**. Saving a type library saves type information to a `.ridl` file. • **Save TLB Button in Type... more (see page 1466)**
Refreshing the Type Library (see page 1467)	To refresh the type library, choose the **Refresh Implementation** icon on the **Type Library Editor toolbar**. Refreshing the type library does the following: • Updates the xxxx_TLB units that contain a Delphi or C++ representation of the Type Library. • Notifies the IDE's module manager to update the implementation files that contain the implementation for the CoClasses, if the type library is associated with a CoClass that was generated by a wizard. On the **Tools ▶ Options ▶ Environment Options ▶ Delphi Options ▶ Type Library** or the **Tools ▶ Options ▶ Environment Options ▶ C++ Options ▶ Type Library** dialog box, you can designate specific instances when the... more (see page 1467)

Registering the Type Library (see page 1467)	To register the type library, click the **Register Type Library** button on the **Type Library Editor toolbar**. Typically, you do not need to explicitly register a type library because it is registered automatically when you register your COM server application (see Registering a COM object (see page 1440)). However, when you create a type library using the **Type Library wizard**, the type library is not associated with a server object. In this case, you can register the type library directly using the toolbar. Registering the type library adds an entry to the Windows registry for the type library.
Deploying Type Libraries (see page 1467)	By default, when you have a type library that was created as part of an Automation server project, the type library is automatically linked into the .DLL, .OCX, or EXE as a resource. You can, however, deploy your application with the type library as a separate .TLB, as Delphi maintains the type library, if you prefer. Historically, type libraries for Automation applications were stored as a separate file with the .TLB extension. Now, typical Automation applications compile the type libraries into the .OCX or .EXE file directly. The operating system expects the type library to be the first resource in... more (see page 1467)
GenTLB.exe (see page 1468)	A new type library generator (GenTLB.exe) has been added to RAD Studio. When you compile or build a project that contains a type library, the type library generator compiles the text-based RIDL file into the binary `.tlb` file. You can use GenTLB.exe in your custom builds. If you use MSBuild from the command line, then MSBuild automatically uses GenTLB.exe. **Examples** To create AppName.tlb from the contents of AppName.ridl:
RIDL File (see page 1468)	A new file type has been added to the COM architecture for 2009 – the RIDL file (Restricted Interface Definition Language). The RIDL file is the storage mechanism that the project uses to persist the type library data to disk. RAD Studio now uses the `.tlb` file as an intermediate file (like a `.res`, `.dcu`, `.obj`, and so forth). This means that you can rebuild the `.tlb` file from the command line (outside the IDE), that `.tlb` files can be edited with a text editor, and the history is stored by the IDE. The Type Library Editor... more (see page 1468)

3.2.2.6.1 Working with Type Libraries: Overview

Type libraries are files that include information about data types, interfaces, member functions, and object classes exposed by a COM object. They provide a way to identify the types of objects and interfaces that are available on a server. For a detailed overview on why and when to use type libraries, see Type libraries (see page 1395).

A type library can contain any and all of the following:

- Information about custom data types such as aliases, enumerations, structures, and unions.
- Descriptions of one or more COM elements, such as an interface, dispinterface, or CoClass. Each of these descriptions is commonly referred to as type information (see page 1449).
- Descriptions of constants and methods defined in external units.
- References to type descriptions from other type libraries.

By including a type library with your COM application or ActiveX library, you make information about the objects in your application available to other applications and programming tools through COM's type library tools and interfaces.

With traditional development tools, you create type libraries by writing scripts in the Interface Definition Language (IDL) or the Object Description Language (ODL), then run that script through a compiler. The Type Library editor automates some of this process, easing the burden of creating and modifying your own type libraries.

In addition, the Type Library Editor now uses an intermediate text file known as a RIDL file (see page 1468). The Code page of the Type Library Editor displays the RIDL code, and you use the Design page to view and change the type library fields (coClass, Methods, Properties, and so forth). When you save or build your project, the RIDL file is converted to a standard binary `.tlb` file. Both the RIDL file and the `.tlb` file are saved in your project.

When you create a COM server of any type (Automation object, remote data module, and so on) using Delphi wizards, the wizard automatically generates a type library for you (although in the case of the COM object wizard, this is optional). Most of the work you do in customizing the generated object starts with the type library, because that is where you define the properties and methods it exposes to clients: you change the interface of the CoClass generated by the wizard, using the

Type Library Editor. The Type Library editor automatically updates the implementation unit for your object, so that all you need do is fill in the bodies of the generated methods.

See Also

Creating Simple COM Servers: Overview (see page 1430)

Overview of COM Technologies (see page 1385)

3.2.2.6.2 Type Library Editor

The **Type Library Editor** enables developers to examine and create type information for COM objects. Using the **Type Library Editor** can greatly simplify the task of developing COM objects by centralizing the tasks of defining interfaces, CoClasses, and types, obtaining GUIDs for new interfaces, associating interfaces with CoClasses, updating implementation units, and so on.

The **Type Library Editor** outputs two types of file that represent the contents of the type library:

Type Library editor files

File	Description
`.ridl` file	The text-based RIDL (Restricted Interface Definition Language) file, introduced for 2009.
`.tlb` file	The binary type library file. By default, you do not need to use this file, because the type library is automatically compiled into the application as a resource. However, you can use this file to explicitly compile the type library into another project or to deploy the type library separately from the .exe or .ocx. For more information, see Opening an existing type library (see page 1460) and Deploying type libraries (see page 1467).
_TLB unit	This unit reflects the contents of the type library for use by your application. It contains all the declarations your application needs to use the elements defined in the type library. Although you can open this file in the code editor, you should never edit it—it is maintained by the **Type Library Editor**, so any changes you make will be overwritten by the **Type Library Editor**. For more details on the contents of this file, see Code generated when you import type library information (see page 1421)

The following topics describe the **Type Library Editor** in greater detail:

- Parts of the Type Library editor (see page 1446)
- Using the Type Library editor (see page 1453)

See Also

Type Library Editor Window

RIDL File (see page 1468)

Toolbar (see page 1447)

Object list pane (see page 1448)

Status Bar (see page 1449)

Pages of Type Information (see page 1449)

3.2.2.6.3 Parts of the Type Library Editor

The main elements of the **Type Library Editor** are described in the following table:

Type Library editor parts

Part	Description
Toolbar (see page 1447)	Includes buttons to add new types, CoClasses, interfaces, and interface members to your type library. The toolbar also includes buttons for refreshing your implementation unit, registering the type library, and saving an IDL file with the information in your type library.
Object list pane (see page 1448)	Displays all the existing elements in the type library. When you click on an item in the object list pane, it displays pages valid for that object.
Status bar (see page 1449)	Displays syntax errors if you try to add invalid types to your type library.
Pages (see page 1449)	Display information about the selected object. Which pages appear here depends on the type of object selected.

See Also

Using the Type Library Editor (see page 1453)

Type Library Editor

3.2.2.6.4 Toolbar

The **Type Library Editor**'s toolbar, located at the top of the Type Library Editor, contains buttons that you click to add new objects into your type library.

The first group of buttons let you add elements to the type library. When you click a toolbar button, the icon for that element appears in the object list pane. You can then customize its attributes in the right pane. Depending on the type of icon you select, different pages of information appear to the right.

The following table lists the elements (see page 1451) you can add to your type library:

Icons Representing Type Library Elements

Icon	Meaning
	An interface description.
	A dispinterface description.
	A CoClass.
	An enumeration.
	An alias.
	A record.
	A union.
	A module.

When you select one of the elements listed above in the object list pane, the second group of buttons displays members that are valid for that element. For example, when you select Interface, the Method and Property icons in the second box become enabled because you can add methods and properties to your interface definition. When you select Enum, the second group of buttons changes to display the Const member, which is the only valid member for Enum type information.

The following table lists the members that can be added to elements in the object list pane:

Icons Representing Members of a Type Library

Icon	Meaning
	A method of the interface, dispinterface, or an entry point in a module.

	A property on an interface or dispinterface.
	A write-only property. (available from the drop-down list on the property button)
	A read-write property. (available from the drop-down list on the property button)
	A read-only property. (available from the drop-down list on the property button)
	A field in a record or union.
	A constant in an enum or a module.

The third group of buttons let you refresh, register, or save a type library as a TLB file, as described in Saving and registering type library information (see page 1465).

Icons Representing Actions You can Perform

Icon	Meaning
	Refresh implementation.
	Register Type Library.
	Save as TLB File.

See Also

Object List Pane (see page 1448)

Status Bar (see page 1449)

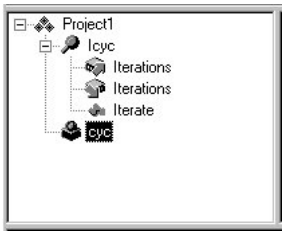
Pages of Type Information (see page 1449)

Elements of Type Libraries (see page 1451)

3.2.2.6.5 Object List Pane

The **Object list** pane displays all the elements of the current type library in a tree view. The root of the tree represents the type library itself, and appears as the following icon:

Descending from the type library node are the elements (see page 1451) in the type library:



When you select any of these elements (including the type library itself), the pages of type information (see page 1449) to the right change to reflect only the relevant information for that element. You can use these pages to edit the definition and properties of the selected element.

You can manipulate the elements in the object list pane by right clicking to get the object list pane context menu. This menu includes commands that let you use the Windows clipboard to move or copy existing elements as well as commands to add new elements or customize the appearance of the **Type Library Editor**.

See Also

Toolbar (see page 1447)

Status Bar (see page 1449)

3.2.2.6.6 Status Bar

When editing or saving a type library, syntax, translation errors, and warnings are listed in the **Message** pane.

For example, if you specify a type that the **Type Library Editor** does not support, you will get a syntax error. For a complete list of types supported by the **Type Library Editor**, see Valid types (see page 1454).

See Also

Object List Pane (see page 1448)

Toolbar (see page 1447)

Pages of Type Information (see page 1449)

3.2.2.6.7 Pages of Type Information

When you select an element (see page 1451) in the object list pane, pages of type information appear in the **Type Library Editor** that are valid for the selected element. Which pages appear depends on the element selected in the object list panel, as follows:

Type Info element	Page of type information	Contents of page
Type library	Attributes	Name, version, and GUID for the type library, as well as information linking the type library to help.
	Uses	List of other type libraries that contain definitions on which this one depends.
	Flags	Flags that determine how other applications can use the type library.
	Text	All definitions and declarations defining the type library itself (see discussion below).
Interface	Attributes	Name, version, and GUID for the interface, the name of the interface from which it descends, and information linking the interface to help.
	Flags	Flags that indicate whether the interface is hidden, dual, Automation-compatible, and/or extensible.
	Text	The definitions and declarations for the Interface (see discussion below).
Dispinterface	Attributes	Name, version, and GUID for the interface, and information linking it to help.
	Flags	Flags that indicate whether the Dispinterface is hidden, dual, and/or extensible.
	Text	The definitions and declarations for the Dispinterface. (see discussion below).
CoClass	Attributes	Name, version, and GUID for the CoClass, and information linking it to help.
	Implements	A List of interfaces that the CoClass implements, as well as their attributes.
	COM+	The attributes of transactional objects, such as the transaction model, call synchronization, just-in-time activation, object pooling, and so on. Also includes the attributes of COM+ event objects.
	Flags	Flags that indicate various attributes of the CoClass, including how clients can create and use instances, whether it is visible to users in a browser, whether it is an ActiveX control, and whether it can be aggregated (act as part of a composite).

	Text	The definitions and declarations for the CoClass (see discussion below).
Enumeration	Attributes	Name, version, and GUID for the enumeration, and information linking it to help.
	Text	The definitions and declarations for the enumerated type (see discussion below).
Alias	Attributes	Name, version, and GUID for the enumeration, the type the alias represents, and information linking it to help.
	Text	The definitions and declarations for the alias (see discussion below).
Record	Attributes	Name, version, and GUID for the record, and information linking it to help.
	Text	The definitions and declarations for the record (see discussion below).
Union	Attributes	Name, version, and GUID for the union, and information linking it to help.
	Text	The definitions and declarations for the union (see discussion below).
Module	Attributes	Name, version, GUID, and associated DLL for the module, and information linking it to help.
	Text	The definitions and declarations for the module (see discussion below).
Method	Attributes	Name, dispatch ID or DLL entry point, and information linking it to help.
	Parameters	Method return type, and a list of all parameters with their types and any modifiers.
	Flags	Flags to indicate how clients can view and use the method, whether this is a default method for the interface, and whether it is replaceable.
	Text	The definitions and declarations for the method (see discussion below).
Property	Attributes	Name, dispatch ID, type of property access method (getter vs. setter), and information linking it to help.
	Parameters	Property access method return type, and a list of all parameters with their types and any modifiers.
	Flags	Flags to indicate how clients can view and use the property, whether this is a default for the interface, whether the property is replaceable, bindable, and so on.
	Text	The definitions and declarations for the property access method (see discussion below).
Const	Attributes	Name, value, type (for module consts), and information linking it to help.
	Flags	Flags to indicate how clients can view and use the constant, whether this represents a default value, whether the constant is bindable, and so on.
	Text	The definitions and declarations for the constant (see discussion below).
Field	Attributes	Name, type, and information linking it to help.
	Flags	Flags to indicate how clients can view and use the field, whether this represents a default value, whether the field is bindable, and so on.
	Text	The definitions and declarations for the field (see discussion below).

You can use each of the pages of type information to view or edit the values it displays. Most of the pages organize the information into a set of controls so that you can type in values or select them from a list without requiring that you know the syntax of the corresponding declarations. This can prevent many small mistakes such as typographic errors when specifying values from a limited set. However, you may find it faster to type in the declarations directly. To do this, use the Text page.

All type library elements have a text page that displays the syntax for the element. This syntax appears in an IDL subset of Microsoft Interface Definition Language, or Delphi. See Using Delphi or IDL syntax (see page 1455) for details. Any changes you make in other pages of the element are reflected on the text page. If you add code directly in the text page, changes are reflected in the other pages of the Type Library editor.

The **Type Library Editor** generates syntax errors if you add identifiers that are currently not supported by the editor; the editor currently supports only those identifiers that relate to type library support (not RPC support or constructs used by the Microsoft

IDL compiler for C++ code generation or marshaling support).

See Also

Object List Pane (see page 1448)

Toolbar (see page 1447)

Status Bar (see page 1449)

Type Library Elements (see page 1451)

3.2.2.6.8 Type Library Elements

The Type Library interface as represented in the Type Library Editor can seem overwhelmingly complicated at first. This is because it represents information about a great number of elements, each of which has its own characteristics. However, many of these characteristics are common to all elements. For example, every element (including the type library itself) has the following:

- **Name**, which is used to describe the element and which is used when referring to the element in code.
- **GUID** (globally unique identifier), which is a unique 128-bit value that COM uses to identify the element. This should always be supplied for the type library itself and for CoClasses and interfaces. It is optional otherwise.
- **Version number**, which distinguishes between multiple versions of the element. This is always optional, but should be provided for CoClasses and interfaces, because some tools can't use them without a version number.
- **Help properties** that link the element to a Help topic. These include a Help String, and Help Context or Help String Context value. The Help Context is used for a traditional Windows Help system where the type library has a stand-alone Help file. The Help String Context is used when help is supplied by a separate DLL instead. The Help Context or Help String Context refers to a Help file or DLL that is specified on the type library's **Attributes** page. This is always optional.

Interfaces

An interface describes the methods (and any properties expressed as `get` and `set` functions) for an object that must be accessed through a virtual function table (vtable). If an interface is flagged as dual, it will inherit from *IDispatch*, and your object can provide both early-bound, vtable access, and runtime binding through OLE automation. By default, the type library flags all interfaces you add as dual.

Interfaces can be assigned members: methods and properties. These appear in the object list pane as children of the interface node. Properties for interfaces are represented by the `get` and `set` methods used to read and write the property's underlying data. They are represented in the tree view using special icons that indicate their purpose.

Special Icons for 'get' and 'set' Methods

	A write (set, put) by value property function.
	A read (get) | write (set, put) | write by reference property function.
	A read (get) property function.

Note: Write by Reference: When a property is specified as Write By Reference, it is passed as a pointer rather than by value. Some applications, such as Visual Basic, use Write By Reference, if it is present, to optimize performance. To pass the property only by reference rather than by value, use the property type *By Reference Only*. To pass the property by reference as well as by value, select Read->Write->Write By Ref

. To invoke this menu, go to the toolbar and select the down-arrow next to the property icon. After you add the properties or methods using the toolbar button or the object list pane context menu, you describe their syntax and attributes by selecting the property or method and using the pages of type information.

The Attributes page lets you give the property or method a name and dispatch ID (so that it can be called using IDispatch). For properties, you also assign a type. The function signature is created using the Parameters page, where you can add, remove,

and rearrange parameters, set their type and any modifiers, and specify function return types.

Note: Members of interfaces that need to raise exceptions should return an HRESULT and specify a return value parameter (PARAM_RETVAL) for the actual return value. Declare these methods using the **safecall** calling convention.

Note that when you assign properties and methods to an interface, they are implicitly assigned to its associated CoClass. This is why the Type Library editor does not let you add properties and methods directly to a CoClass.

Dispinterfaces

Interfaces are more commonly used than dispinterfaces to describe the properties and methods of an object. Dispinterfaces are only accessible through dynamic binding, while interfaces can have static binding through a vtable.

You can add methods and properties to dispinterfaces in the same way you add them to interfaces. However, when you create a property for a dispinterface, you can't specify a function kind or parameter types.

CoClasses

A CoClass describes a unique COM object that implements one or more interfaces. When defining a CoClass, you must specify which implemented interface is the default for the object, and optionally, which dispinterface is the default source for events. Note that you do not add properties or methods to a CoClass in the Type Library editor. Properties and methods are exposed to clients by interfaces, which are associated with the CoClass using the Implements page.

Type definitions

Enumerations, aliases, records, and unions all declare types that can then be used elsewhere in the type library.

Enums consist of a list of constants, each of which must be numeric. Numeric input is usually an integer in decimal or hexadecimal format. The base value is zero by default. You can add constants to your enumeration by selecting the enumeration in the object list pane and clicking the Const button on the toolbar or selecting **New ▶ Const** command from the object list pane context menu.

Note: It is strongly recommended that you provide help strings for your enumerations to make their meaning clearer. The following is a sample entry of an enumeration type for a mouse button and includes a help string for each enumeration element.

```
mbLeft = 0 [helpstring 'mbLeft'];
mbRight = 1 [helpstring 'mbRight'];
mbMiddle = 3 [helpstring 'mbMiddle'];
typedef enum TxMouseButton
{
[helpstring("mbLeft")]
mbLeft = 0,
[helpstring("mbRight)]
mbRight = 1.
[helpstring("mbMiddle)]
mbMiddle = 2
} TxMouseButton;
```

An alias creates an alias (type definition) for a type. You can use the alias to define types that you want to use in other type info such as records or unions. Associate the alias with the underlying type definition by setting the Type attribute on the Attributes page.

A record consists of a list of structure members or fields. A union is a record with only a variant part. Like a record, a union consists of a list of structure members or fields. However, unlike the members of records, each member of a union occupies the same physical address, so that only one logical value can be stored.

Add the fields to a record or union by selecting it in the object list pane and clicking the field button in the toolbar or right clicking and choosing field from the object list pane context menu. Each field has a name and a type, which you assign by selecting the field and assigning values using the Attributes page. Records and unions can be defined with an optional tag.

Members can be of any built-in type, or you can specify a type using alias before you define the record.

Modules

A module defines a group of functions, typically a set of DLL entry points. You define a module by

- Specifying a DLL that it represents on the attributes page.
- Adding methods and constants using the toolbar or the object list pane context menu. For each method or constant, you must then define its attributes by selecting the it in the object list pane and setting the values on the Attributes page.

For module methods, you must assign a name and DLL entry point using the attributes page. Declare the function's parameters and return type using the parameters page.

For module constants, use the Attributes page to specify a name, type, and value.

Note: The Type Library Editor

does not generate any declarations or implementation related to a module. The specified DLL must be created as a separate project.

See Also

Pages of Type Information (see page 1449)

Toolbar (see page 1447)

Type Library Editor

Type Library Explorer (Read-Only)

3.2.2.6.9 Using the Type Library Editor

Using the type library editor, you can create new type libraries or edit existing ones. Typically, an application developer uses a wizard to create the objects that are exposed in the type library, letting Delphi generate the type library automatically. Then, the automatically-generated type library is opened in the Type Library editor so that the interfaces can be defined (or modified), type definitions added, and so on.

However, even if you are not using a wizard to define the objects, you can use the Type Library editor to define a new type library. In this case, you must create any implementation classes yourself, because the Type Library editor does not generate code for CoClasses that were not associated with a type library by a wizard.

The editor supports a subset of valid types (see page 1454) in a type library

The following topics describe how to:

- Create a new type library (see page 1460)
- Open an existing type library (see page 1460)
- Add an interface to the type library (see page 1461)
- Modify an interface (see page 1461)
- Add properties and methods to the type library (see page 1462)
- Add a CoClass to the type library (see page 1463)
- Add an interface to a CoClass (see page 1463)
- Add an enumeration to the type library (see page 1464)
- Add an alias to the type library (see page 1464)
- Add a record or union to the type library (see page 1464)
- Add a module to the type library (see page 1465)
- Save and register type library information (see page 1465)

See Also

Type Libraries (see page 1395)

Parts of the Type Library Editor (see page 1446)

3.2.2.6.10 Valid Types

In the Type Library editor, you use different type identifiers, depending on whether you are working in IDL or Delphi. Specify the language you want to use in the Environment options dialog.

The following types are valid in a type library for COM development. The Automation compatible column specifies whether the type can be used by an interface that has its Automation or Dispinterface flag checked. These are the types that COM can marshal via the type library automatically.

Delphi type	IDL type	variant type	Automation compatible	Description
Smallint	short	VT_I2	Yes	2-byte signed integer
Integer	long	VT_I4	Yes	4-byte signed integer
Single	single	VT_R4	Yes	4-byte real
Double	double	VT_R8	Yes	8-byte real
Currency	CURRENCY	VT_CY	Yes	currency
TDateTime	DATE	VT_DATE	Yes	date
WideString	BSTR	VT_BSTR	Yes	binary string
IDispatch	IDispatch	VT_DISPATCH	Yes	pointer to IDispatch interface
SCODE	SCODE	VT_ERROR	Yes	Ole Error Code
WordBool	VARIANT_BOOL	VT_BOOL	Yes	True = -1, False = 0
OleVariant	VARIANT	VT_VARIANT	Yes	Ole Variant
IUnknown	IUnknown	VT_UNKNOWN	Yes	pointer to IUnknown interface
Shortint	byte	VT_I1	No	1 byte signed integer
Byte	unsigned char	VT_UI1	Yes	1 byte unsigned integer
Word	unsigned short	VT_UI2	Yes*	2 byte unsigned integer
LongWord	unsigned long	VT_UI4	Yes*	4 byte unsigned integer
Int64	__int64	VT_I8	No	8 byte signed integer
Largeuint	uint64	VT_UI8	No	8 byte unsigned integer
SYSINT	int	VT_INT	Yes*	system dependent integer (Win32=Integer)
SYSUINT	unsigned int	VT_UINT	Yes*	system dependent unsigned integer
HResult	HRESULT	VT_HRESULT	No	32 bit error code
Pointer		VT_PTR -> VT_VOID	No	untyped pointer
SafeArray	SAFEARRAY	VT_SAFEARRAY	No	OLE Safe Array
PAnsiChar	LPSTR	VT_LPSTR	No	pointer to Char
PChar	LPWSTR	VT_LPWSTR	No	pointer to WideChar
PWideChar	LPWSTR	VT_LPWSTR	No	pointer to WideChar

- Word, LongWord, SYSINT, and SYSUINT may be Automation-compatible with some applications.

See safe arrays (see page 1455) for more information about the SAFEARRAY Variant type.

Note: The Byte (VT_UI1) is Automation-compatible, but is not allowed in a Variant or OleVariant since many Automation servers do not handle this value correctly.

Besides these IDL types, any interfaces and types defined in the library or defined in referenced libraries can be used in a type library definition.

The Type Library editor stores type information in the generated type library (.TLB) file in binary form.

If a parameter type is specified as a Pointer type, the Type Library editor usually translates that type into a variable parameter. When the type library is saved, the variable parameter's associated ElemDesc's IDL flags are marked IDL_FIN or IDL_FOUT.

Often, ElemDesc IDL flags are not marked by IDL_FIN or IDL_FOUT when the type is preceded with a Pointer. Or, in the case of dispinterfaces, IDL flags are not typically used. In these cases, you may see a comment next to the variable identifier such as {IDL_None} or {IDL_In}. These comments are used when saving a type library to correctly mark the IDL flags.

3.2.2.6.11 SafeArrays

COM requires that arrays be passed via a special data type known as a SafeArray. You can create and destroy SafeArrays by calling special COM functions to do so, and all elements within a SafeArray must be valid automation-compatible types (see page 1454). The Delphi compiler has built-in knowledge of COM SafeArrays and automatically calls the COM API to create, copy, and destroy SafeArrays.

In the **Type Library Editor**, a *SafeArray* must specify the type of its elements. For example, the following line from the text page declares a method with a parameter that is a *SafeArray* with an element type of Integer:

```
procedure HighLightLines(Lines: SafeArray of Integer);
HRESULT _stdcall HighlightLines(SAFEARRAY(long) Lines);
```

Note: Although you must specify the element type when declaring a *SafeArray* type in the Type Library Editor

, the declaration in the generated _TLB unit does not indicate the element type.

See Also

Valid Types (see page 1454)

3.2.2.6.12 Using Object Pascal or IDL Syntax

The Code page of the Type Library editor displays your type information in RIDL format (Restricted Interface Definition Language). The format resembles Microsoft's IDL sytnax. However, RIDL supports only a subset of IDL. Like Delphi applications in general, identifiers in type libraries are case insensitive. Identifiers can be as many as 255 characters long, and for maximum portability should begin with a letter or an underscore (_).

Attribute specifications

Types declared in a RIDL file may contain attributes. Attribute specifications appear enclosed in square brackets and separated by commas. Each attribute specification consists of an attribute name followed (if appropriate) by a value.

The following table lists the attribute names and their corresponding values.

Attribute syntax

Attribute name	Example	Applies to
aggregatable	[aggregatable]	typeinfo
appobject	[appobject]	CoClass typeinfo
bindable	[bindable]	members except CoClass members

control	[control]	type library, typeinfo
custom	[custom '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}' 0]	anything
default	[default]	CoClass members
defaultbind	[defaultbind]	members except CoClass members
defaultcollection	[defaultcollection]	members except CoClass members
dispid	[dispid]	members except CoClass members
displaybind	[displaybind]	members except CoClass members
dllname	[dllname 'Helper.dll']	module typeinfo
dual	[dual]	interface typeinfo
helpfile	[helpfile 'c:\help\myhelp.hlp']	type library
helpstringdll	[helpstringdll 'c:\help\myhelp.dll']	type library
helpcontext	[helpcontext 2005]	anything except CoClass members and parameters
helpstring	[helpstring 'payroll interface']	anything except CoClass members and parameters
helpstringcontext	[helpstringcontext $17]	anything except CoClass members and parameters
hidden	[hidden]	anything except parameters
lcid	[lcid $324]	type library
licensed	[licensed]	type library, CoClass typeinfo
nonbrowsable	[nonbrowsable]	members except CoClass members
nonextensible	[nonextensible]	interface typeinfo
oleautomation	[oleautomation]	interface typeinfo
predeclid	[predeclid]	typeinfo
propget	[propget]	members except CoClass members
propput	[propput]	members except CoClass members
propputref	[propputref]	members except CoClass members
public	[public]	alias typeinfo
readonly	[readonly]	members except CoClass members
replaceable	[replaceable]	anything except CoClass members and parameters
requestedit	[requestedit]	members except CoClass members
restricted	[restricted]	anything except parameters
source	[source]	all members
uidefault	[uidefault]	members except CoClass members
usesgetlasterror	[usesgetlasterror]	members except CoClass members
uuid	[uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}']	type library, typeinfo (required)

vararg	[vararg]	members except CoClass members
version	[version 1.1]	type library, typeinfo

Interface syntax

The Delphi syntax for declaring interface type information has the form

```
interfacename = interface[(baseinterface)] [attributes]
functionlist
[propertymethodlist]
end;
```

For example, the following text declares an interface with two methods and one property:

```
Interface1 = interface (IDispatch)
  [uuid '{7B5687A1-F4E9-11D1-92A8-00C04F8C8FC4}', version 1.0]
  function Calculate(optional seed:Integer=0): Integer;
  procedure Reset;
  procedure PutRange(Range: Integer) [propput, dispid $00000005]; stdcall;
  function GetRange: Integer;[propget, dispid $00000005]; stdcall;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',version 1.0]
interface Interface1 :IDispatch
{
  long Calculate([in, optional, defaultvalue(0)] long seed);
  void Reset(void);
  [propput, id(0x00000005)] void _stdcall PutRange([in] long Value);
  [propput, id(0x00000005)] void _stdcall getRange([out, retval] long *Value);
};
```

Dispatch interface syntax

The Delphi syntax for declaring dispinterface type information has the form

```
dispinterfacename = dispinterface [attributes]
functionlist
[propertylist]
end;
```

For example, the following text declares a dispinterface with the same methods and property as the previous interface:

```
MyDispObj = dispinterface
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
  version 1.0,
  helpstring 'dispatch interface for MyObj'
  function Calculate(seed:Integer): Integer [dispid 1];
  procedure Reset [dispid 2];
  property Range: Integer [dispid 3];
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
  version 1.0,
  helpstring "dispatch interface for MyObj"
dispinterface Interface1
{
  methods:
    [id(1)] int Calculate([in] int seed);
    [id(2)] void Reset(void);
  properties:
    [id(3)] int Value;
};
```

CoClass syntax

The Delphi syntax for declaring CoClass type information has the form

```
classname = coclass(interfacename[interfaceattributes], ...); [attributes];
```

For example, the following text declares a coclass for the interface *IMyInt* and dispinterface *DmyInt*:

```
myapp = coclass(IMyInt [source], DMyInt);
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  version 1.0,
  helpstring 'A class',
  appobject]
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
  version 1.0,
  helpstring "A class",
  appobject]
coclass myapp
{
  methods:
  [source] interface IMyInt);
  dispinterface DMyInt;
};
```

Enum syntax

The Delphi syntax for declaring Enum type information has the form

```
enumname = ([attributes] enumlist);
```

For example, the following text declares an enumerated type with three values:

```
location = ([uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
         helpstring 'location of booth']
  Inside = 1 [helpstring 'Inside the pavillion'];
  Outside = 2 [helpstring 'Outside the pavillion'];
  Offsite = 3 [helpstring 'Not near the pavillion'];);
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring "location of booth"]
typedef enum
{
  [helpstring "Inside the pavillion"] Inside = 1,
  [helpstring "Outside the pavillion"] Outside = 2,
  [helpstring "Not near the pavillion"] Offsite = 3
} location;
```

Alias syntax

The Delphi syntax for declaring Alias type information has the form

```
aliasname = basetype[attributes];
```

For example, the following text declares DWORD as an alias for integer:

```
DWORD = Integer [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'];
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}'] typedef long DWORD;
```

Record syntax

The Delphi syntax for declaring Record type information has the form

```
recordname = record [attributes] fieldlist end;
```

For example, the following text declares a record:

```
Tasks = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                helpstring 'Task description']
  ID: Integer;
  StartDate: TDate;
  EndDate: TDate;
  Ownername: WideString;
  Subtasks: safearray of Integer;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring "Task description"]
typedef struct
{
  long ID;
  DATE StartDate;
  DATE EndDate;
  BSTR Ownername;
  SAFEARRAY (int) Subtasks;
} Tasks;
```

Union syntax

The Delphi syntax for declaring Union type information has the form

```
unionname = record [attributes]
case Integer of
  0: field1;
  1: field2;
  ...
end;
```

For example, the following text declares a union:

```
MyUnion = record [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                  helpstring "item description"]
case Integer of
  0: (Name: WideString);
  1: (ID: Integer);
  3: (Value: Double);
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        helpstring "item description"]
typedef union
{
  BSTR Name;
  long ID;
  double Value;
  } MyUnion;
```

Module syntax

The Delphi syntax for declaring Module type information has the form

```
modulename = module constants entrypoints end;
```

For example, the following text declares the type information for a module:

```
MyModule = module [uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
                   dllname 'circle.dll']
  PI: Double = 3.14159;
  function area(radius: Double): Double [ entry 1 ]; stdcall;
  function circumference(radius: Double): Double [ entry 2 ]; stdcall;
end;
```

The corresponding syntax in Microsoft IDL is

```
[uuid '{2MD36ABF-90E3-11D1-AA75-02C04FB73F42}',
        dllname("circle.dll")]
module MyModule
{
  double PI = 3.14159;
  [entry(1)] double _stdcall area([in] double radius);
  [entry(2)] double _stdcall circumference([in] double radius);
};
```

See Also

Valid Types (see page 1454)

3.2.2.6.13 Creating a New Type Library

Use the **Type Library Wizard** to create a type library that is independent of a particular COM object. For example, you might want to define a type library that contains type definitions that you use in several other type libraries. You can then create a type library of basic definitions and add it to the `uses` page of other type libraries.

You can also use the wizard to create a type library for an object that is not yet implemented. Once the type library contains the interface definition, you can use the COM object wizard (see page 1431) to generate a CoClass and implementation.

To create a new type library

1. If you have a project open, click **File** ▸ **Close All**.
2. Choose **File** ▸ **New** ▸ **Other** to open the New Items dialog box.
3. Choose the **ActiveX** folder under either **Delphi Projects** or **C++Builder Projects**.
4. Select the **Type Library** icon in the right pane.
5. Choose **OK**. The wizard creates an empty RIDL file and opens it in the Design page of the Type Library Editor(in the Code Editor window).
6. Enter a name for the type library in the **Name**field.
7. Continue by adding elements to your type library by using the toolbar (see page 1447) buttons.

See Also

Opening an Existing Type Library (see page 1460)

Using the Type Library Editor (see page 1453)

Working with Type Libraries (see page 1441)

RIDL File (see page 1468)

3.2.2.6.14 Opening an Existing Type Library

When you use the wizards to create an Automation object, COM object, transactional object, or a remote data module, a type library is automatically created with an implementation unit. Starting with the 2009 product, the wizards create type libraries in a text-based format (RIDL files). However, you can still open type libraries in `.tlb` format that are associated with previous versions or other products (servers) that are available on your system. The Type Library Explorer enables you to examine `.tlb` files.

To open a legacy type library (.tlb file)

1. Choose **File** ▸ **Open**.

2. In the Open dialog box, set the File Type to type library.
3. Navigate to the desired `.tlb` file and choose Open.

The Type Library Explorer opens in the Code Editor. You can view the type library but you cannot edit it in the Explorer.

To open a type library created with a COM wizard in the 2009 product

1. In the Project Manager, locate the `.ridl` file associated with your project.
2. Double-click the `.ridl` file.

The Type Library Editor opens in the Code Editor window.

Alternatively, you can choose **View** ▶ **Type Library**. On the Design page of the Type Library Editor, you can add interfaces, CoClasses, and other elements of the type library such as enumerations, properties, and methods.

Note: Changes you make to any type library information with the Type Library Editor

can be automatically reflected in the associated implementation class. If you want to review the changes before they are added, be sure that the Apply Updates (see page 1466) dialog is checked on. It is on by default and can be changed in the setting "Display updates before refreshing" on the **Tools** ▶ **Options** ▶ **Delphi Options** ▶ **Type Library** page.

Tip: When writing client applications, you typically do not need to open the type library. You only need the *Project*_TLB unit that was generated by the Import Component wizard or by TLIBIMP, not the type library itself. You can add this file directly to a client project, or, if the type library is registered on your system, you can use the Import Type Library dialog (Component ->Import Type Library

).

See Also

Creating a New Type Library (see page 1460)

3.2.2.6.15 Adding an Interface to the Type Library

To add an interface

1. On the toolbar, click on the interface icon. An interface is added to the object list pane prompting you to add a name.
2. Type a name for the interface.

The new interface contains default attributes that you can modify as needed.

You can add properties (represented by getter/setter functions) and methods to suit the purpose of the interface.

See Also

Toolbar (see page 1447)

Adding Properties and Methods to the Type Library (see page 1462)

Modifying an Interface Using the Type Library (see page 1461)

3.2.2.6.16 Modifying an Interface Using the Type Library

There are several ways to modify an interface or dispinterface once it is created.

- You can change the interface's attributes using the page of type information that contains the information you want to change. Select the interface in the **Object List** pane and then use the controls on the appropriate page of type information. For example, you may want to change the parent interface using the attributes page, or use the flags page to change whether or not it is a dual interface.
- You can edit the interface declaration directly by selecting the interface in the object list pane and then editing the declarations on the **Text** page.
- You can Add properties and methods to the interface (see page 1462).

- You can modify the properties and methods already in your interface by changing their type information.
- You can associate it with a CoClass by selecting the CoClass in the object list pane, right-clicking on the Implements page, and choosing Insert Interface.

If the interface is associated with a CoClass that was generated by a wizard, you can tell the **Type Library Editor** to apply your changes to the implementation file by clicking the **Refresh** button on the toolbar. If you have the Apply Updates (see page 1466) dialog enabled, the **Type Library Editor** notifies you before updating the sources and warns you of potential problems. For example, if you rename an event interface by mistake, you may get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name. As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

You also get a TODO comment in your source file immediately above it.

Warning: If you ignore this warning and TODO comment, the code will not compile.

See Also

Adding an Interface to the Type Library (see page 1461)

Adding Properties and Methods to the Type Library (see page 1462)

3.2.2.6.17 Adding Properties and Methods to the Type Library

To add properties or methods to an interface or dispinterface

1. Select the interface, and choose either a **property** or **method** icon from the toolbar. If you are adding a property, you can click directly on the property icon to create a read/write property (with both a getter and a setter), or click the down arrow to display a menu of property types. The property access method members or method member is added to the object list pane, prompting you to add a name.
2. Type a name for the member.

The new member contains default settings on its attributes, parameters, and flags pages that you can modify to suit the member. For example, you will probably want to assign a type to a property on the attributes page. If you are adding a method, you will probably want to specify its parameters on the parameters page.

As an alternate approach, you can add properties and methods by typing directly into the text page using Delphi or IDL syntax. For example, if you are working in Delphi syntax, you can type the following property declarations into the text page of an interface:

```
Interface1 = interface(IDispatch)
  [ uuid '{5FD36EEF-70E5-11D1-AA62-00C04FB16F42}',
    version 1.0,
    dual,
    oleautomation ]
 function AutoSelect: Integer [propget, dispid $00000002]; safecall; // Add this
 function AutoSize: WordBool [propget, dispid $00000001]; safecall; // And this
 procedure AutoSize(Value: WordBool) [propput, dispid $00000001]; safecall; // And this
end;
```

If you are working in IDL, you can add the same declarations as follows:

```
[
   uuid(5FD36EEF-70E5-11D1-AA62-00C04FB16F42),
   version(1.0),
   dual,
   oleautomation
]
interface Interface1: IDispatch
{ // Add everything between the curly braces
[propget, id(0x00000002)]
   HRESULT _stdcall AutoSelect([out, retval] long Value );
   [propget, id(0x00000003)]
```

```
    HRESULT _stdcall AutoSize([out, retval] VARIANT_BOOL Value );
    [propput, id(0x00000003)]
    HRESULT _stdcall AutoSize([in] VARIANT_BOOL Value );
};
```

After you have added members to an interface using the interface text page, the members appear as separate items in the object list pane, each with its own attributes, flags, and parameters pages. You can modify each new property or method by selecting it in the object list pane and using these pages, or by making edits directly in the text page.

If the interface is associated with a CoClass that was generated by a wizard, you can tell the **Type Library Editor** to apply your changes to the implementation file by clicking the **Refresh** button on the toolbar. The **Type Library Editor** adds new methods to your implementation class to reflect the new members. You can then locate the new methods in implementation unit's source code and fill in their bodies to complete the implementation.

If you have the Apply Updates (see page 1466) dialog enabled, the **Type Library Editor** notifies you of all changes before updating the sources and warns you of potential problems.

See Also

Modifying an Interface Using the Type Library (see page 1461)

Adding an Interface to the Type Library (see page 1461)

Toolbar (see page 1447)

3.2.2.6.18 Adding a CoClass to the Type Library

The easiest way to add a CoClass to your project is to choose **File** ▸ **New** ▸ **Other** from the main menu in the IDE and use the appropriate wizard (see page 1397) on the ActiveX page of the **New Items** dialog. The advantage to this approach is that, in addition to adding the CoClass and its interface to the type library, the wizard adds an implementation unit and updates the project file to include the new implementation unit in its uses clause.

If you are not using a wizard, however, you can create a CoClass by clicking the CoClass icon on the toolbar and then specifying its attributes. You will probably want to give the new CoClass a name (on the Attributes page), and may want to use the Flags page to indicate information such as whether the CoClass is an application object, whether it represents an ActiveX control, and so on.

Note: When you add a CoClass to a type library using the toolbar instead of a wizard, you must generate the implementation for the CoClass yourself and update it by hand every time you change an element on one of the CoClass' interfaces.

You can't add members directly to a CoClass. Instead, you implicitly add members when you add an interface to the CoClass (see page 1463).

See Also

Adding an Interface to the Type Library (see page 1461)

Code Generated by Wizards (see page 1398)

3.2.2.6.19 Adding an Interface to a CoClass

CoClasses are defined by the interfaces they present to clients. While you can add any number of properties and methods to the implementation class of a CoClass, clients can only see those properties and methods that are exposed by interfaces associated with the CoClass.

To associate an interface with a CoClass, right-click in the **Implements** page for the class and choose Insert Interface to display a list of interfaces from which you can choose. The list includes interfaces that are defined in the current type library and those defined in any type libraries that the current type library references. Choose an interface you want the class to implement. The

interface is added to the page with its GUID and other attributes.

If the CoClass was generated by a wizard, the **Type Library Editor** automatically updates the implementation class to include skeletal methods for the methods (including property access methods) of any interfaces you add this way.If you have the Apply Updates (see page 1466) dialog enabled, the **Type Library Editor** notifies you before updating the sources and warns you of potential problems.

See Also

Adding an Interface to the Type Library (see page 1461)

Adding a CoClass to the Type Library (see page 1463)

3.2.2.6.20 Adding an Enumeration to the Type Library

To add enumerations to a type library

1. On the toolbar, click on the enum icon. An enum type is added to the **Object List** pane .
2. Type a name for the enumeration. The new enum is empty and contains default attributes in its attributes page for you to modify.

Add values to the enum by right clicking the enum and selecting the **New** ▸ **Const** button . Then, select each enumerated value and assign it a name (and possibly a value) using the attributes page.

Once you have added an enumeration, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the enumeration as the type for a property or parameter.

See Also

Adding an Alias to the Type Library (see page 1464)

Adding a Record or Union to the Type Library (see page 1464)

Adding a Module to the Type Library (see page 1465)

3.2.2.6.21 Adding an Alias to the Type Library

To add an alias to a type library

1. On the toolbar, click on the alias icon. An alias type is added to the object list pane .
2. Type a name for the alias. By default, the new alias stands for a Long Integer type. Use the **Attributes** page to change this to the type you want the alias to represent.

Once you have added an alias, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the alias as the type for a property or parameter.

See Also

Adding an Enumeration to the Type Library (see page 1464)

Adding a Record or Union to the Type Library (see page 1464)

Adding a Module to the Type Library (see page 1465)

3.2.2.6.22 Adding a Record or Union to the Type Library

To add a record or union to a type library

1. On the toolbar, click on the record icon or the union icon. The selected type element is added to the object list pane .
2. Type a name for the record or union. At this point, the new record or union contains no fields.

3. With the record or union selected in the object list pane, click on the field icon in the toolbar. Specify the field's name and type, using the **Attributes** page.
4. Repeat step 3 for as many fields as you need.

Once you have defined the record or union, the new type is available for use by the type library or any other type library that references it from its uses page. For example, you can use the record or union as the type for a property or parameter.

See Also

Adding an Enumeration to the Type Library (see page 1464)

Adding an Alias to the Type Library (see page 1464)

Adding a Module to the Type Library (see page 1465)

3.2.2.6.23 Adding a Module to the Type Library

To add a module to a type library

1. On the toolbar, click on the module icon. The selected module is added to the object list pane .
2. Type a name for the module.
3. On the **Attributes** page, specify the name of the DLL whose entry points the Module represents.
4. Add any methods from the DLL you specified in step 3 by clicking on the **Method** icon in the toolbar and then using the attributes pages to describe the method.
5. Add any constants you want the module to define by clicking on the **Const** icon on the toolbar. For each constant, specify a name, type, and value.

See Also

Adding an Enumeration to the Type Library (see page 1464)

Adding an Alias to the Type Library (see page 1464)

3.2.2.6.24 Saving and Registering Type Library Information

After modifying your type library, you'll want to save and register the type library information.

Saving the type library automatically updates:

- Both the text-based type library file (`.ridl` extension) and the binary type library file (`.tlb` extension).
- The *Project*_TLB unit that represents its contents
- The implementation code for any CoClasses that were generated by a wizard.

Note: The type library is stored as separate text (.RIDL) and binary (.TLB) files, but is also linked into the server (.EXE, DLL, or .OCX).

The **Type Library Editor** gives you toolbar buttons for storing your type library information:

- Refresh (see page 1467) updates the type library units in memory only.
- Register (see page 1467) adds an entry for the type library in your system's Windows registry. This is done automatically when the associated server is itself registered.
- Save as TLB (see page 1466) saves the type library as a `.tlb` file.

All the above methods perform syntax checking. When you refresh, register, or save the type library, Delphi automatically updates the implementation unit of any CoClasses that were created using a wizard.

See Also

Type Library Editor Window

RIDL File (see page 1468)

Registering the Type Library (see page 1467)

Saving a Type Library (see page 1466)

3.2.2.6.25 Apply Updates Dialog

The **Apply Updates** dialog appears when you refresh, register, or save the type library if you have selected **Display updates before refreshing** in the **Tools** ▸ **Options** ▸ **Type Library** page (which is not checked off by default).

Without this option, the **Type Library Editor** automatically updates the sources of the associated object when you make changes in the editor. With this option, you have a chance to veto the proposed changes when you attempt to refresh, save, or register the type library.

The **Apply Updates** dialog will warn you about potential errors, and will insert TODO comments in your source file. For example, if you rename an event by mistake, you will get a warning in your source file that looks like this:

```
Because of the presence of instance variables in your implementation file,
Delphi was not able to update the file to reflect the change in your event
interface name. As Delphi has updated the type library for you, however, you
must update the implementation file by hand.
```

You will also get a TODO comment in your source file immediately above it.

Note: If you ignore this warning and TODO comment, the code will not compile.

See Also

Refreshing the Type Library (see page 1467)

Registering the Type Library (see page 1467)

3.2.2.6.26 Saving a Type Library

For 2009, RAD Studio stores type libraries as separate text-based files (RIDL, or Restricted Interface Definition Language); The RIDL file, which represents the TLB file in the Editor, is also linked into the server (.EXE, DLL, or .OCX). When you build your project, the RIDL file is saved on disk as both a `.tlb` file as well as a `.ridl` file.

There are three ways to save a type library.

- **File>Save Command:** To save the project and the type library, choose **File** ▸ **Save**. Saving a type library saves type information to a `.ridl` file.
- **Save TLB Button in Type Library Editor:** To save the type library as a `.tlb` file, click **Save as TLB** in the toolbar (see page 1447) of the Type Library Editor. Saving as a TLB creates a binary type library file (`.tlb` extension) derived from the `.ridl` file in your type library.
- **Build Your Project or Run GenTLB:** Both building your project and running GenTLB.exe also save the type library as a `.tlb` file.

You might encounter unexpected messages from COM when you save a type library as a TLB file (such as when you perform a project build). No syntax or validity checks are performed while you are working with the `.ridl` file or when you merely save the project.

See Also

Apply Updates Dialog (see page 1466)

Refreshing the Type Library (see page 1467)

Registering the Type Library (see page 1467)

Type Library Editor Window

RIDL File (see page 1468)

3.2.2.6.27 Refreshing the Type Library

To refresh the type library, choose the **Refresh Implementation** icon on the **Type Library Editor toolbar**.

Refreshing the type library does the following:

- Updates the xxxx_TLB units that contain a Delphi or C++ representation of the Type Library.
- Notifies the IDE's module manager to update the implementation files that contain the implementation for the CoClasses, if the type library is associated with a CoClass that was generated by a wizard.

On the **Tools ▶ Options ▶ Environment Options ▶ Delphi Options ▶ Type Library** or the **Tools ▶ Options ▶ Environment Options ▶ C++ Options ▶ Type Library** dialog box, you can designate specific instances when the Type Library Editor is to refresh the type library implementation. You can select from the following instances:

- When renaming the type library.
- When .ridl file is saved.

Note: If you have renamed items in the type library, refreshing the implementation may create duplicate entries. In this case, you must move your code to the correct entry and delete any duplicates. Similarly, if you delete items in the type library, refreshing the implementation does not remove them from CoClasses (under the assumption that you are merely removing them from visibility to clients). You must delete these items manually in the implementation unit if they are no longer needed.

See Also

Apply Updates Dialog (see page 1466)

Saving a Type Library (see page 1466)

Registering the Type Library (see page 1467)

3.2.2.6.28 Registering the Type Library

To register the type library, click the **Register Type Library** button on the **Type Library Editor toolbar**.

Typically, you do not need to explicitly register a type library because it is registered automatically when you register your COM server application (see Registering a COM object (see page 1440)). However, when you create a type library using the **Type Library wizard**, the type library is not associated with a server object. In this case, you can register the type library directly using the toolbar.

Registering the type library adds an entry to the Windows registry for the type library.

See Also

Apply Updates Dialog (see page 1466)

Saving a Type Library (see page 1466)

Refreshing the Type Library (see page 1467)

3.2.2.6.29 Deploying Type Libraries

By default, when you have a type library that was created as part of an Automation server project, the type library is automatically linked into the .DLL, .OCX, or EXE as a resource.

You can, however, deploy your application with the type library as a separate .TLB, as Delphi maintains the type library, if you prefer.

Historically, type libraries for Automation applications were stored as a separate file with the .TLB extension. Now, typical Automation applications compile the type libraries into the .OCX or .EXE file directly. The operating system expects the type library to be the first resource in the executable (.DLL, .OCX, or .EXE) file.

When you make type libraries other than the primary project type library available to application developers, the type libraries can be in any of the following forms:

- Stand-alone binary files. The .TLB file output by the Type Library editor is a binary file.
- A resource. This resource should have the type TYPELIB and an integer ID. If you choose to build type libraries with a resource compiler, it must be declared in the resource (.RC) file as follows:

```
1 typelib mylib1.tlb
2 typelib mylib2.tlb
```

See Also

Deploying Applications (see page 1947)

3.2.2.6.30 GenTLB.exe

A new type library generator (GenTLB.exe) has been added to RAD Studio. When you compile or build a project that contains a type library, the type library generator compiles the text-based RIDL file into the binary `.tlb` file.

You can use GenTLB.exe in your custom builds. If you use MSBuild from the command line, then MSBuild automatically uses GenTLB.exe.

Examples

To create AppName.tlb from the contents of AppName.ridl:

```
C:\> gentlb  AppName.ridl
```

To create a type library based on a differently named .ridl file, use the -T option to specify the name of the type library. To create AppName.tlb from the contents of model.ridl:

```
C:\> gentlb -TAppName.tlb  model.ridl
```

See Also

Type Library Editor

RIDL File (see page 1468)

Object List Pane (see page 1448)

Toolbar (see page 1447)

Pages of Type Information (see page 1449)

Using the COM Wizards (see page 79)

3.2.2.6.31 RIDL File

A new file type has been added to the COM architecture for 2009 – the RIDL file (Restricted Interface Definition Language). The RIDL file is the storage mechanism that the project uses to persist the type library data to disk. RAD Studio now uses the `.tlb` file as an intermediate file (like a `.res`, `.dcu`, `.obj`, and so forth). This means that you can rebuild the `.tlb` file from the command line (outside the IDE), that `.tlb` files can be edited with a text editor, and the history is stored by the IDE.

The Type Library Editor now works on top of a .text file (the RIDL file) instead of the binary `.tlb`.

This change means that:

- You no longer have to check in the binary `.tlb` file, because the build-system can generate it from the latest .RIDL file.
- Multiple programmers can work on the type library at the same time and then simply merge their changes to the text RIDL file. [Binary files cannot be merged, requiring that the last person always overwrites previous changes.]

The RIDL format offers more flexibility for certain constructs than the Type Library Editor. For example, with a RIDL file you can add and edit custom data on any type, function, or member. You can describe complex types such as safe arrays of safe arrays. You can easily perform diff's of RIDL files and gain insight into the evolution of the model exposed by a server over time.

During a build, the RIDL file is compiled using the new type library generator, GenTLB.exe, to generate the binary `.tlb` file.

See Also

Type Library Editor

GenTLB.exe (see page 1468)

Object List Pane (see page 1448)

Toolbar (see page 1447)

Pages of Type Information (see page 1449)

Using the COM Wizards (see page 79)

3.2.3 Developing Database Applications

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Contains the Developer's Guide topics for programming database applications.

Topics

Name	Description
Working with ADO components (see page 1470)	
Connecting to databases (see page 1495)	
Creating multi-tiered applications (see page 1510)	
Creating reports with Rave Reports (see page 1551)	
Designing database applications (see page 1556)	
Understanding datasets (see page 1573)	
Using the Borland Database Engine (see page 1637)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.
Using client datasets (see page 1700)	
Using data controls (see page 1743)	
Using decision support components (see page 1779)	
Using provider components (see page 1805)	
Using dbExpress Components (see page 1821)	
Using XML in database applications (see page 1840)	
Working with field components (see page 1849)	

3.2.3.1 Working with ADO components

Topics

Name	Description
ADO Connection Events (see page 1474)	In addition to the usual events that occur for all database connection components, *TADOConnection* generates a number of additional events that occur during normal usage.
Accessing the Connection Object (see page 1475)	Use the ConnectionObject property of *TADOConnection* to access the underlying ADO connection object. Using this reference it is possible to access properties and call methods of the underlying ADO Connection object. Using the underlying ADO Connection object requires a good working knowledge of ADO objects in general and the ADO Connection object in particular. It is not recommended that you use the Connection object unless you are familiar with Connection object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO Connection objects.
Accessing the Connection's Datasets (see page 1475)	Like other database connection components, you can access the datasets associated with the connection (see page 1505) using the *DataSets* and *DataSetCount* properties. However, *dbGo* also includes TADOCommand objects, which are not datasets, but which maintain a similar relationship to the connection component. You can use the Commands and CommandCount properties of *TADOConnection* to access the associated ADO command objects in the same way you use the *DataSets* and *DataSetCount* properties to access the associated datasets. Unlike *DataSets* and *DataSetCount*, which only list active datasets, *Commands* and *CommandCount* provide references to all *TADOCommand* components associated with the connection component. *Commands* is a... more (see page 1475)
Applying the Batch Updates to Base Tables (see page 1476)	Apply pending data changes that have not yet been applied or canceled by calling the UpdateBatch method. Rows that have been changed and are applied have their changes put into the base tables on which the recordset is based. A cached row marked for deletion causes the corresponding base table row to be deleted. A record insertion (exists in the cache but not the base table) is added to the base table. Modified rows cause the columns in the corresponding rows in the base tables to be changed to the new column values in the cache. Used alone with no... more (see page 1476)
Canceling Batch Updates (see page 1476)	Cancel pending data changes that have not yet been canceled or applied by calling the CancelBatch method. When you cancel pending batch updates, field values on rows that have been changed revert to the values that existed prior to the last call to *CancelBatch* or *UpdateBatch*, if either has been called, or prior to the current pending batch of changes. Used alone with no parameter, *CancelBatch* cancels all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *CancelBatch*. If any value except *arAll* is passed, only a subset of the pending changes are canceled.... more (see page 1476)
Canceling Commands (see page 1477)	If you are executing the command asynchronously, then after calling Execute you can abort the execution by calling the Cancel method:
Connecting to ADO Data Stores (see page 1477)	dbGo applications use Microsoft ActiveX Data Objects (ADO) 2.1 to interact with an OLE DB provider that connects to a data store and accesses its data. One of the items a data store can represent is a database. An ADO-based application requires that ADO 2.1 be installed on the client computer. ADO and OLE DB is supplied by Microsoft and installed with Windows. An ADO provider represents one of a number of types of access, from native OLE DB drivers to ODBC drivers. These drivers must be installed on the client computer. OLE DB drivers for... more (see page 1477)
Connecting to a Data Store Using TADOConnection (see page 1478)	One or more ADO dataset and command components can share a single connection to a data store by using TADOConnection. To do so, associated dataset and command components with the connection component through their *Connection* properties. At design-time, select the desired connection component from the drop-down list for the *Connection* property in the **Object Inspector**. At runtime, assign the reference to the *Connection* property. For example, the following line associates a *TADODataSet* component with a *TADOConnection* component.
Connecting an ADO Dataset to a Data Store (see page 1479)	ADO datasets can connect to an ADO data store either collectively or individually. When connecting datasets collectively, set the Connection property of each dataset to a TADOConnection component. Each dataset then uses the ADO connection component's connection.

Controlling Timeouts (see page 1480)	You can control the amount of time that can elapse before attempted commands and connections are considered failed and are aborted using the ConnectionTimeout and CommandTimeout properties. ConnectionTimeout specifies the amount of time, in seconds, before an attempt to connect to the data store times out. If the connection does not successfully compile prior to expiration of the time specified in ConnectionTimeout, the connection attempt is canceled:
Fetching Records Asynchronously (see page 1481)	Unlike other datasets, ADO datasets can fetch their data asynchronously. This allows your application to continue performing other tasks while the dataset populates itself with data from the data store. To control whether the dataset fetches data asynchronously, if it fetches data at all, use the ExecuteOptions property. *ExecuteOptions* governs how the dataset fetches its records when you call *Open* or set *Active* to *True*. If the dataset represents a query or stored procedure that does not return any records, ExecuteOptions governs how the query or stored procedure is executed when you call *ExecSQL* or *ExecProc*. *ExecuteOptions* is... more (see page 1481)
Filtering Multiple Rows Based On Update Status (see page 1481)	Filter a recordset to show only those rows that belong to a group of rows with the same update status using the FilterGroup property. Set *FilterGroup* to the *TFilterGroup* constant that represents the update status of rows to display. A value of *fgNone* (the default value for this property) specifies that no filtering is applied and all rows are visible regardless of update status (except rows marked for deletion). The example below causes only pending batch update rows to be visible.
Filtering Records Based On Bookmarks (see page 1482)	ADO datasets support the common dataset feature of using bookmarks to mark and return to specific records (see page 1599). Also like other datasets, ADO datasets let you use filters to limit the available records in the dataset (see page 1631). ADO datasets provide an additional feature that combines these two common dataset features: the ability to filter on a set of records identified by bookmarks.
Fine-tuning a Connection (see page 1483)	One advantage of using *TADOConnection* for establishing the connection to a data store instead of simply supplying a connection string for your ADO command and dataset components, is that it provides a greater degree of control over the conditions and attributes of the connection. The following topics describe the properties you can use to fine-tune the connection: • Forcing asynchronous connections (see page 1483) • Controlling time-outs (see page 1480) • Indicating the types of operations the connection supports (see page 1485) • Specifying whether the connection automatically initiates transactions (see page 1489)
Forcing Asynchronous Connections (see page 1483)	Use the ConnectOptions property to force the connection to be asynchronous. Asynchronous connections allow your application to continue processing without waiting for the connection to be completely opened. By default, *ConnectionOptions* is set to *coConnectUnspecified* which allows the server to decide the best type of connection. To explicitly make the connection asynchronous, set *ConnectOptions* to *coAsyncConnect*. The example routines below enable and disable asynchronous connections in the specified connection component:

Handling Command Parameters (see page 1484)	There are two ways in which a *TADOCommand* object may use parameters: • The *CommandText* property can specify a query that includes parameters. Working with parameterized queries in *TADOCommand* works like using a parameterized query (see page 1617) in an ADO dataset. • The *CommandText* property can specify a stored procedure that uses parameters. Stored procedure parameters (see page 1625) work much the same using *TADOCommand* as with an ADO dataset. There are two ways to supply parameter values when working with *TADOCommand*: you can supply them when you call the Execute method, or you can specify them ahead of time using the Parameters property. The *Execute... more (see page 1484)*
Indicating the Types of Operations the Connection Supports (see page 1485)	ADO connections are established using a specific mode, similar to the mode you use when opening a file. The connection mode determines the permissions available to the connection, and hence the types of operations (such as reading and writing) that can be performed using that connection. Use the Mode property to indicate the connection mode. The possible values are listed in the following table: ***ADO connection modes***
Inspecting the Update Status of Individual Rows (see page 1485)	Determine the update status of a given row by making it current and then inspecting the RecordStatus property of the ADO data component. *RecordStatus* reflects the update status of the current row and only that row.
Loading Data from and Saving Data to Files (see page 1486)	The data retrieved via an ADO dataset component can be saved to a file for later retrieval on the same or a different computer. The data is saved in one of two proprietary formats: ADTG or XML. These two file formats are the only formats supported by ADO. However, both formats are not necessarily supported in all versions of ADO. Consult the ADO documentation for the version you are using to determine what save file formats are supported. Save the data to a file using the SaveToFile method. *SaveToFile* takes two parameters, the name of the file to which data... more (see page 1486)
Opening the Dataset in Batch Update Mode (see page 1487)	To open an ADO dataset in batch update mode, it must meet these criteria: 1. The component's CursorType property must be *ctKeySet* (the default property value) or *ctStatic*. 2. The LockType property must be *ltBatchOptimistic*. 3. The command must be a SELECT query. Before activating the dataset component, set the *CursorType* and *LockType* properties as indicated above. Assign a SELECT statement to the component's CommandText property (for *TADODataSet*) or the SQL property (for *TADOQuery*). For *TADOStoredProc* components, set the *ProcedureName* to the name of a stored procedure that returns a result set. These properties can be set at design-time... more (see page 1487)
Overview of ADO Components (see page 1487)	The ADO page of the **Tool palette** hosts the *dbGo* components. These components let you connect to an ADO data store, execute commands, and retrieve data from tables in databases using the ADO framework. They require ADO 2.1 (or higher) to be installed on the host computer. Additionally, client software for the target database system (such as Microsoft SQL Server) must be installed, as well as an OLE DB driver or ODBC driver specific to the particular database system. Most *dbGo* components have direct counterparts in the components available for other data access mechanisms: a database connection component (... more (see page 1487)

Retrieving Result Sets with Commands (see page 1488)	Unlike *TADOQuery* components, which use different methods to execute depending on whether they return a result set, *TADOCommand* always uses the Execute command to execute the command, regardless of whether it returns a result set. When the command returns a result set, *Execute* returns an interface to the ADO _RecordSet interface. The most convenient way to work with this interface is to assign it to the RecordSet property of an ADO dataset. For example, the following code uses *TADOCommand* (*ADOCommand1*) to execute a SELECT query, which returns a result set. This result set is then assigned to the... more (see page 1488)
Specifying the Command (see page 1489)	Specify commands for a *TADOCommand* component using the CommandText property. Like *TADODataSet*, *TADOCommand* lets you specify the command in different ways, depending on the CommandType property. Possible values for *CommandType* include: *cmdText* (used if the command is an SQL statement), *cmdTable* (if it is a table name), and *cmdStoredProc* (if the command is the name of a stored procedure). At design-time, select the appropriate command type from the list in the **Object Inspector**. At runtime, assign a value of type *TCommandType* to the *CommandType* property.
Specifying Whether the Connection Automatically Initiates Transactions (see page 1489)	Use the Attributes property to control the connection component's use of retaining commits and retaining aborts. When the connection component uses retaining commits, then every time your application commits a transaction, a new transaction is automatically started. When the connection component uses retaining aborts, then every time your application rolls back a transaction, a new transaction is automatically started. *Attributes* is a set that can contain one, both, or neither of the constants *xaCommitRetaining* and *xaAbortRetaining*. When *Attributes* contains *xaCommitRetaining*, the connection uses retaining commits. When *Attributes* contains *xaAbortRetaining*, it uses retaining aborts. Check whether either retaining... more (see page 1489)
Using ADO datasets (see page 1490)	ADO dataset components encapsulate the ADO Recordset object. They inherit the common dataset capabilities described in Understanding Datasets (see page 1632), using ADO to provide the implementation. In order to use an ADO dataset, you must familiarize yourself with these common features. In addition to the common dataset features, all ADO datasets add properties, events, and methods for the following: • Connecting to an ADO datastore (see page 1479) • Accessing the underlying Recordset object (see page 1494) • Filtering records based on bookmarks (see page 1482) • Fetching records asynchronously (see page 1481) • Performing batch updates (caching updates) (see page 1491) • Using files on disk to store data (see page 1486) There are four ADO datasets: • *TADOTable*, a table-type dataset (see page 1620) that represents... more (see page 1490)
Using Batch Updates (see page 1491)	One approach for caching updates is to connect the ADO dataset to a client dataset using a dataset provider. This approach is discussed in Using a client dataset to cache updates (see page 1731). However, ADO dataset components provide their own support for cached updates, which they call batch updates. The following table lists the correspondences between caching updates using a client dataset and using the batch updates features: ***Comparison of ADO and client dataset cached updates***
Using Command Objects (see page 1492)	In the ADO environment, commands are textual representations of provider-specific action requests. Typically, they are Data Definition Language (DDL) and Data Manipulation Language (DML) SQL statements. The language used in commands is provider-specific, but usually compliant with the SQL-92 standard for the SQL language. Although you can always execute commands using *TADOQuery*, you may not want the overhead of using a dataset component, especially if the command does not return a result set. As an alternative, you can use the TADOCommand component, which is a lighter-weight object designed to execute commands, one command at a time. *TADOCommand* is intended... more (see page 1492)

Using TADODataSet (see page 1492)	TADODataSet is a general-purpose dataset (see page 1632) for working with data from an ADO data store. Unlike the other ADO dataset components, *TADODataSet* is not a table-type (see page 1620), query-type (see page 1618), or stored procedure-type (see page 1618) dataset. Instead, it can function as any of these types: • Like a table-type dataset, *TADODataSet* lets you represent all of the rows and columns of a single database table. To use it in this way, set the CommandType property to *cmdTable* and the CommandText property to the name of the table. *TADODataSet* supports table-type tasks such as • Assigning indexes (see page 1609) to sort records or form the basis of record-based searches.... more (see page 1492)
Using the Execute Method (see page 1493)	Before *TADOCommand* can execute its command, it must have a valid connection to a data store. This is established just as with an ADO dataset. See Connecting an ADO dataset to a data store (see page 1479) for details. To execute the command, call the Execute method. *Execute* is an overloaded method that lets you choose the most appropriate way to execute the command. For commands that do not require any parameters and for which you do not need to know how many records were affected, call *Execute* without any parameters:
Working with ADO Components (see page 1494)	The *dbGo* components provide data access through the ADO framework. ADO, (Microsoft ActiveX Data Objects) is a set of COM objects that access data through an OLE DB provider. The *dbGo* components encapsulate these ADO objects in the Delphi database architecture. The ADO layer of an ADO-based application consists of Microsoft ADO 2.1, an OLE DB provider or ODBC driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these must... more (see page 1494)
Working with Record Sets (see page 1494)	The Recordset property provides direct access to the ADO recordset object underlying the dataset component. Using this object, it is possible to access properties and call methods of the recordset object from an application. Use of *Recordset* to directly access the underlying ADO recordset object requires a good working knowledge of ADO objects in general and the ADO recordset object in specific. Using the recordset object directly is not recommended unless you are familiar with recordset object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO recordset objects. The RecordSetState property indicates the current state... more (see page 1494)

3.2.3.1.1 ADO Connection Events

In addition to the usual events that occur for all database connection components, *TADOConnection* generates a number of additional events that occur during normal usage.

Events when establishing a connection

In addition to the *BeforeConnect* and *AfterConnect* events that are common to all database connection components, *TADOConnection* also generates an *OnWillConnect* and *OnConnectComplete* event when establishing a connection. These events occur after the *BeforeConnect* event.

- OnWillConnect occurs before the ADO provider establishes a connection. It lets you make last minute changes to the connection string, provide a user name and password if you are handling your own login support, force an asynchronous connection, or even cancel the connection before it is opened.
- OnConnectComplete occurs after the connection is opened. Because *TADOConnection* can represent asynchronous connections, you should use *OnConnectComplete*, which occurs after the connection is opened or has failed due to an error condition, instead of the *AfterConnect* event, which occurs after the connection component instructs the ADO provider to open a connection, but not necessarily after the connection is opened.

Events when disconnecting

In addition to the *BeforeDisconnect* and *AfterDisconnect* events common to all database connection components,

TADOConnection also generates an OnDisconnect event after closing a connection. *OnDisconnect* occurs after the connection is closed but before any associated datasets are closed and before the *AfterDisconnect* event.

Events when managing transactions

The ADO connection component provides a number of events for detecting when transaction-related processes have been completed. These events indicate when a transaction process initiated by a *BeginTrans*, *CommitTrans*, and *RollbackTrans* method has been successfully completed at the data store.

- The OnBeginTransComplete event occurs when the data store has successfully started a transaction after a call to the BeginTrans method.
- The OnCommitTransComplete event occurs after a transaction is successfully committed due to a call to CommitTrans.
- The OnRollbackTransComplete event occurs after a transaction is successfully aborted due to a call to *RollbackTrans*.

Other events

ADO connection components introduce two additional events you can use to respond to notifications from the underlying ADO connection object:

- The OnExecuteComplete event occurs after the connection component executes a command on the data store (for example, after calling the *Execute* method). *OnExecuteComplete* indicates whether the execution was successful.
- The OnInfoMessage event occurs when the underlying connection object provides detailed information after an operation is completed. The *OnInfoMessage* event handler receives the interface to an ADO Error object that contains the detailed information and a status code indicating whether the operation was successful.

See Also

Connecting to Databases (see page 1506)

Connecting to a Data Store Using TADOConnection (see page 1478)

Connecting to a Data Store Using TADOConnection (see page 1478)

Accessing the Connection's Datasets (see page 1475)

3.2.3.1.2 Accessing the Connection Object

Use the ConnectionObject property of *TADOConnection* to access the underlying ADO connection object. Using this reference it is possible to access properties and call methods of the underlying ADO Connection object.

Using the underlying ADO Connection object requires a good working knowledge of ADO objects in general and the ADO Connection object in particular. It is not recommended that you use the Connection object unless you are familiar with Connection object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO Connection objects.

3.2.3.1.3 Accessing the Connection's Datasets

Like other database connection components, you can access the datasets associated with the connection (see page 1505) using the *DataSets* and *DataSetCount* properties. However, *dbGo* also includes TADOCommand objects, which are not datasets, but which maintain a similar relationship to the connection component.

You can use the Commands and CommandCount properties of *TADOConnection* to access the associated ADO command objects in the same way you use the *DataSets* and *DataSetCount* properties to access the associated datasets. Unlike *DataSets* and *DataSetCount*, which only list active datasets, *Commands* and *CommandCount* provide references to all *TADOCommand* components associated with the connection component.

Commands is a zero-based array of references to ADO command components. *CommandCount* provides a total count of all of the commands listed in *Commands*. You can use these properties together to iterate through all the commands that use a

connection component, as illustrated in the following code:

```
var
  i: Integer
begin
  for i := 0 to (ADOConnection1.CommandCount - 1) do
    ADOConnection1.Commands[i].Execute;
end;
for (int i = 0; i < ADOConnection2->CommandCount; i++)
  ADOConnection2->Commands[i]->Execute();
```

See Also

Connecting to Databases (see page 1506)

Connecting to a Data Store Using TADOConnection (see page 1478)

Connecting to a Data Store Using TADOConnection (see page 1478)

ADO Connection Events (see page 1474)

3.2.3.1.4 Applying the Batch Updates to Base Tables

Apply pending data changes that have not yet been applied or canceled by calling the UpdateBatch method. Rows that have been changed and are applied have their changes put into the base tables on which the recordset is based. A cached row marked for deletion causes the corresponding base table row to be deleted. A record insertion (exists in the cache but not the base table) is added to the base table. Modified rows cause the columns in the corresponding rows in the base tables to be changed to the new column values in the cache.

Used alone with no parameter, *UpdateBatch* applies all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *UpdateBatch*. If any value except *arAll* is passed, only a subset of the pending changes are applied. Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be applied. The example below applies only the currently active row to be applied:

```
ADODataSet1.UpdateBatch(arCurrent);
ADODataSet1->UpdateBatch(arCurrent);
```

See Also

Opening the Dataset in Batch Update Mode (see page 1487)

Inspecting the Update Status of Individual Rows (see page 1485)

Filtering Multiple Rows Based On Update Status (see page 1481)

Canceling Batch Updates (see page 1476)

3.2.3.1.5 Canceling Batch Updates

Cancel pending data changes that have not yet been canceled or applied by calling the CancelBatch method. When you cancel pending batch updates, field values on rows that have been changed revert to the values that existed prior to the last call to *CancelBatch* or *UpdateBatch*, if either has been called, or prior to the current pending batch of changes.

Used alone with no parameter, *CancelBatch* cancels all pending updates. A *TAffectRecords* value can optionally be passed as the parameter for *CancelBatch*. If any value except *arAll* is passed, only a subset of the pending changes are canceled. Passing *arAll* is the same as passing no parameter at all and causes all pending updates to be canceled. The example below cancels all pending changes:

```
ADODataSet1.CancelBatch;
ADODataSet1->CancelBatch(arAll);
```

See Also

Opening the Dataset in Batch Update Mode (see page 1487)

Inspecting the Update Status of Individual Rows (see page 1485)

Filtering Multiple Rows Based On Update Status (see page 1481)

Applying the Batch Updates to Base Tables (see page 1476)

3.2.3.1.6 Canceling Commands

If you are executing the command asynchronously, then after calling Execute you can abort the execution by calling the Cancel method:

```
procedure TDataForm.ExecuteButtonClick(Sender: TObject);
begin
  ADOCommand1.Execute;
end;
procedure TDataForm.CancelButtonClick(Sender: TObject);
begin
  ADOCommand1.Cancel;
end;
void __fastcall TDataForm::ExecuteButtonClick(TObject *Sender)
{
  ADOCommand1->Execute();
}
void __fastcall TDataForm::CancelButtonClick(TObject *Sender)
{
  ADOCommand1->Cancel();
}
```

The *Cancel* method only has an effect if there is a command pending and it was executed asynchronously (*eoAsynchExecute* is in the *ExecuteOptions* parameter of the *Execute* method). A command is said to be pending if the *Execute* method has been called but the command has not yet been completed or timed out.

A command times out if it is not completed or canceled before the number of seconds specified in the CommandTimeout property expire. By default, commands time out after 30 seconds.

See Also

Specifying the Command (see page 1489)

Using the Execute Method (see page 1493)

Retrieving Result Sets with Commands (see page 1488)

Handling Command Parameters (see page 1484)

3.2.3.1.7 Connecting to ADO Data Stores

dbGo applications use Microsoft ActiveX Data Objects (ADO) 2.1 to interact with an OLE DB provider that connects to a data store and accesses its data. One of the items a data store can represent is a database. An ADO-based application requires that ADO 2.1 be installed on the client computer. ADO and OLE DB is supplied by Microsoft and installed with Windows.

An ADO provider represents one of a number of types of access, from native OLE DB drivers to ODBC drivers. These drivers must be installed on the client computer. OLE DB drivers for various database systems are supplied by the database vendor or by a third-party. If the application uses an SQL database, such as Microsoft SQL Server or Oracle, the client software for that database system must also be installed on the client computer. Client software is supplied by the database vendor and installed from the database systems CD (or disk).

To connect your application with the data store (see page 1478), use an ADO connection component (TADOConnection). Configure the ADO connection component to use one of the available ADO providers. Although *TADOConnection* is not strictly required, because ADO command and dataset components can establish connections directly using their *ConnectionString* property, you can use *TADOConnection* to share a single connection among several ADO components. This can reduce resource consumption, and allows you to create transactions that span multiple datasets.

Like other database connection components (see page 1506), *TADOConnection* provides support for

- Controlling connections (see page 1497)
- Controlling server login (see page 1507)
- Managing transactions (see page 1498)
- Working with associated datasets (see page 1505)
- Sending commands to the server (see page 1502)
- Obtaining metadata (see page 1501)

In addition to these features that are common to all database connection components, *TADOConnection* provides its own support for

- A wide range of options you can use to fine-tune the connection (see page 1483).
- The ability to list the command objects that use the connection (see page 1475).
- Additional events (see page 1474) when performing common tasks.

See Also

TADOConnection

Using TADODataSet (see page 1492)

Using Command Objects (see page 1492)

3.2.3.1.8 Connecting to a Data Store Using TADOConnection

One or more ADO dataset and command components can share a single connection to a data store by using TADOConnection. To do so, associated dataset and command components with the connection component through their *Connection* properties. At design-time, select the desired connection component from the drop-down list for the *Connection* property in the **Object Inspector**. At runtime, assign the reference to the *Connection* property. For example, the following line associates a *TADODataSet* component with a *TADOConnection* component.

```
ADODataSet1.Connection := ADOConnection1;
ADODataSet1->Connection = ADOConnection1;
```

The connection component represents an ADO connection object (see page 1475). Before you can use the connection object to establish a connection, you must identify the data store to which you want to connect. Typically, you provide information using the ConnectionString property. *ConnectionString* is a semicolon delimited string that lists one or more named connection parameters. These parameters identify the data store by specifying either the name of a file that contains the connection information or the name of an ADO provider and a reference identifying the data store. Use the following, predefined parameter names to supply this information:

Connection parameters

Parameter	Description
Provider	The name of a local ADO provider to use for the connection.
Data Source	The name of the data store.
File name	The name of a file containing connection information.

Remote Provider	The name of an ADO provider that resides on a remote machine.
Remote Server	The name of the remote server when using a remote provider.

Thus, a typical value of *ConnectionString* has the form

```
Provider=MSDASQL.1;Data Source=MQIS
```

Note: The connection parameters in *ConnectionString* do not need to include the *Provider* or *Remote Provider* parameter if you specify an ADO provider using the Provider property. Similarly, you do not need to specify the *Data Source* parameter if you use the DefaultDatabase property.

In addition, to the parameters listed above, *ConnectionString* can include any connection parameters peculiar to the specific ADO provider you are using. These additional connection parameters can include user ID and password if you want to hardcode the login information.

At design-time, you can use the Connection String Editor to build a connection string by selecting connection elements (like the provider and server) from lists. Click the ellipsis button for the *ConnectionString* property in the **Object Inspector** to launch the Connection String Editor, which is an ActiveX property editor supplied by ADO.

Once you have specified the *ConnectionString* property (and, optionally, the *Provider* property), you can use the ADO connection component to connect to (see page 1509) or disconnect from (see page 1497) the ADO data store, although you may first want to use other properties to fine-tune the connection (see page 1483). When connecting to or disconnecting from the data store, *TADOConnection* lets you respond to a few additional events (see page 1474) beyond those common to all database connection components..

Note: If you do not explicitly activate the connection by setting the connection component's *Connected* property to *True*, it automatically establishes the connection when the first dataset component is opened or the first time you use an ADO command component to execute a command.

See Also

Connection

Connection

Connection

Connection

Connection

Connecting to Databases (see page 1506)

Fine-tuning a Connection (see page 1483)

Accessing the Connection's Datasets (see page 1475)

ADO Connection Events (see page 1474)

3.2.3.1.9 Connecting an ADO Dataset to a Data Store

ADO datasets can connect to an ADO data store either collectively or individually.

When connecting datasets collectively, set the Connection property of each dataset to a TADOConnection component. Each dataset then uses the ADO connection component's connection.

```
ADODataSet1.Connection := ADOConnection1;
ADODataSet2.Connection := ADOConnection1;
...
ADODataSet1->Connection = ADOConnection1;
```

```
ADODataSet2->Connection = ADOConnection1;
...
```

Among the advantages of connecting datasets collectively are:

- The datasets share the connection object's attributes.
- Only one connection need be set up: that of the *TADOConnection*.
- The datasets can participate in transactions.

For more information on using *TADOConnection* see Connecting to ADO data stores (see page 1477).

When connecting datasets individually, set the ConnectionString property of each dataset. Each dataset that uses *ConnectionString* establishes its own connection to the data store, independent of any other dataset connection in the application.

The *ConnectionString* property of ADO datasets works the same way as the *ConnectionString* property of *TADOConnection*: it is a set of semicolon-delimited connection parameters such as the following:

```
ADODataSet1.ConnectionString := "Provider=YourProvider;Password=SecretWord;" +
  "User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;" +
  "Initial Catalog=Employee";
ADODataSet1->ConnectionString = "Provider=YourProvider;Password=SecretWord;";
ADODataSet1->ConnectionString += "User ID=JaneDoe;SERVER=PURGATORY";
ADODataSet1->ConnectionString += "UID=JaneDoe;PWD=SecretWord;"
ADODataSet1->ConnectionString += "Initial Catalog=Employee";
```

At design time you can use the Connection String Editor to help you build the connection string. For more information about connection strings, see Connecting to a data store using TADOConnection (see page 1478).

See Also

Filtering Records Based On Bookmarks (see page 1482)

Working with Record Sets (see page 1494)

Using Batch Updates (see page 1491)

Loading Data from and Saving Data to Files (see page 1486)

Fetching Records Asynchronously (see page 1481)

3.2.3.1.10 Controlling Timeouts

You can control the amount of time that can elapse before attempted commands and connections are considered failed and are aborted using the ConnectionTimeout and CommandTimeout properties.

ConnectionTimeout specifies the amount of time, in seconds, before an attempt to connect to the data store times out. If the connection does not successfully compile prior to expiration of the time specified in ConnectionTimeout, the connection attempt is canceled:

```
with ADOConnection1 do begin
  ConnectionTimeout := 10 {seconds};
  Open;
end;
  ADOConnection1->ConnectionTimeout = 10; // seconds
  ADOConnection1->Open();
```

CommandTimeout specifies the amount of time, in seconds, before an attempted command times out. If a command initiated by a call to the Execute method does not successfully complete prior to expiration of the time specified in CommandTimeout, the command is canceled and ADO generates an exception:

```
with ADOConnection1 do begin
  CommandTimeout := 10 {seconds};
  Execute("DROP TABLE Employee1997", cmdText, []);
```

```
end;
  ADOConnection1->ConnectionTimeout = 10;
  ADOConnection1->Execute("DROP TABLE Employee1997", cmdText, TExecuteOptions());
```

See Also

Specifying Whether the Connection Automatically Initiates Transactions (see page 1489)

Indicating the Types of Operations the Connection Supports (see page 1485)

3.2.3.1.11 Fetching Records Asynchronously

Unlike other datasets, ADO datasets can fetch their data asynchronously. This allows your application to continue performing other tasks while the dataset populates itself with data from the data store.

To control whether the dataset fetches data asynchronously, if it fetches data at all, use the ExecuteOptions property. *ExecuteOptions* governs how the dataset fetches its records when you call *Open* or set *Active* to *True*. If the dataset represents a query or stored procedure that does not return any records, ExecuteOptions governs how the query or stored procedure is executed when you call *ExecSQL* or *ExecProc*.

ExecuteOptions is a set that includes zero or more of the following values:

Execution options for ADO datasets

Execute Option	Meaning
eoAsyncExecute	The command or data fetch operation is executed asynchronously.
eoAsyncFetch	The dataset first fetches the number of records specified by the *CacheSize* property synchronously, then fetches any remaining rows asynchronously.
eoAsyncFetchNonBlocking	Asynchronous data fetches or command execution do not block the current thread of execution.
eoExecuteNoRecords	A command or stored procedure that does not return data. If any rows are retrieved, they are discarded and not returned.

See Also

Connecting an ADO Dataset to a Data Store (see page 1479)

Working with Record Sets (see page 1494)

Using Batch Updates (see page 1491)

Loading Data from and Saving Data to Files (see page 1486)

Filtering Records Based On Bookmarks (see page 1482)

3.2.3.1.12 Filtering Multiple Rows Based On Update Status

Filter a recordset to show only those rows that belong to a group of rows with the same update status using the FilterGroup property. Set *FilterGroup* to the *TFilterGroup* constant that represents the update status of rows to display. A value of *fgNone* (the default value for this property) specifies that no filtering is applied and all rows are visible regardless of update status (except rows marked for deletion). The example below causes only pending batch update rows to be visible.

```
FilterGroup := fgPendingRecords;
Filtered := True;
FilterGroup = fgPendingRecords;
Filtered = true;
```

Note: For the *FilterGroup* property to have an effect, the ADO dataset component's Filtered property must be set to *True*.

See Also

Opening the Dataset in Batch Update Mode (see page 1487)

Inspecting the Update Status of Individual Rows (see page 1485)

Applying the Batch Updates to Base Tables (see page 1476)

Canceling Batch Updates (see page 1476)

3.2.3.1.13 Filtering Records Based On Bookmarks

ADO datasets support the common dataset feature of using bookmarks to mark and return to specific records (see page 1599). Also like other datasets, ADO datasets let you use filters to limit the available records in the dataset (see page 1631). ADO datasets provide an additional feature that combines these two common dataset features: the ability to filter on a set of records identified by bookmarks.

To filter on a set of bookmarks

1. Use the Bookmark method to mark the records you want to include in the filtered dataset.
2. Call the FilterOnBookmarks method to filter the dataset so that only the bookmarked records appear.

This process is illustrated below:

```
procedure TForm1.Button1Click(Sender: TObject);
var
BM1, BM2: TBookmarkStr;
begin
with ADODataSet1 do begin
BM1 := Bookmark;
    BMList.Add(Pointer(BM1));
MoveBy(3);
BM2 := Bookmark;
    BMList.Add(Pointer(BM2));
    FilterOnBookmarks([BM1, BM2]);
end;
end;
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TBookmarkStr BM1;
TBookmarkStr BM2;
BM1 = ADODataSet1->Bookmark;
  BMList->Add(BM1);
ADODataSet1->MoveBy(3);
BM2 = ADODataSet1->Bookmark;
  BMList->Add(BM2);
  ADODataSet1->FilterOnBookmarks(ARRAYOFCONST((BM1,BM2)));
}
```

Note that the example above also adds the bookmarks to a list object named BMList. This is necessary so that the application can later free the bookmarks when they are no longer needed.

See Also

Connecting an ADO Dataset to a Data Store (see page 1479)

Working with Record Sets (see page 1494)

Using Batch Updates (see page 1491)

Loading Data from and Saving Data to Files (see page 1486)

Fetching Records Asynchronously (see page 1481)

3.2.3.1.14 Fine-tuning a Connection

One advantage of using *TADOConnection* for establishing the connection to a data store instead of simply supplying a connection string for your ADO command and dataset components, is that it provides a greater degree of control over the conditions and attributes of the connection.

The following topics describe the properties you can use to fine-tune the connection:

- Forcing asynchronous connections (see page 1483)
- Controlling time-outs (see page 1480)
- Indicating the types of operations the connection supports (see page 1485)
- Specifying whether the connection automatically initiates transactions (see page 1489)

See Also

Connecting to Databases (see page 1506)

Connecting to a Data Store Using TADOConnection (see page 1478)

Accessing the Connection's Datasets (see page 1475)

ADO Connection Events (see page 1474)

3.2.3.1.15 Forcing Asynchronous Connections

Use the ConnectOptions property to force the connection to be asynchronous. Asynchronous connections allow your application to continue processing without waiting for the connection to be completely opened.

By default, *ConnectionOptions* is set to *coConnectUnspecified* which allows the server to decide the best type of connection. To explicitly make the connection asynchronous, set *ConnectOptions* to *coAsyncConnect*.

The example routines below enable and disable asynchronous connections in the specified connection component:

```
procedure TForm1.AsyncConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coAsyncConnect;
    Open;
  end;
end;
procedure TForm1.ServerChoiceConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coConnectUnspecified;
    Open;
  end;
end;
void __fastcall TForm1::AsyncConnectButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  ADOConnection1->ConnectOptions = coAsyncConnect;
  ADOConnection1->Open();
}
void __fastcall TForm1::ServerChoiceConnectButtonClick(TObject *Sender)
{
  ADOConnection1->Close();
  ADOConnection1->ConnectOptions = coConnectUnspecified;
  ADOConnection1->Open();
}
```

See Also

Controlling Timeouts (see page 1480)

Specifying Whether the Connection Automatically Initiates Transactions (see page 1489)

Indicating the Types of Operations the Connection Supports (see page 1485)

3.2.3.1.16 Handling Command Parameters

There are two ways in which a *TADOCommand* object may use parameters:

- The *CommandText* property can specify a query that includes parameters. Working with parameterized queries in *TADOCommand* works like using a parameterized query (see page 1617) in an ADO dataset.
- The *CommandText* property can specify a stored procedure that uses parameters. Stored procedure parameters (see page 1625) work much the same using *TADOCommand* as with an ADO dataset.

There are two ways to supply parameter values when working with *TADOCommand*: you can supply them when you call the Execute method, or you can specify them ahead of time using the Parameters property.

The *Execute* method is overloaded to include versions that take a set of parameter values as a Variant array. This is useful when you want to supply parameter values quickly without the overhead of setting up the *Parameters* property:

```
ADOCommand1.Execute(VarArrayOf([Edit1.Text, Date]));
Variant Values[2];
Values[0] = Edit1->Text;
Values[1] = Date();
ADOCommand1.Execute(VarArrayOf(Values,1));
```

When working with stored procedures that return output parameters, you must use the *Parameters* property instead. Even if you do not need to read output parameters, you may prefer to use the *Parameters* property, which lets you supply parameters at design time and lets you work with *TADOCommand* properties in the same way you work with the parameters on datasets.

When you set the *CommandText* property, the *Parameters* property is automatically updated to reflect the parameters in the query or those used by the stored procedure. At design-time, you can use the Parameter Editor to access parameters, by clicking the ellipsis button for the *Parameters* property in the **Object Inspector**. At runtime, use properties and methods of *TParameter* to set (or get) the values of each parameter.

```
with ADOCommand1 do begin
  CommandText := 'INSERT INTO Talley ' +
    '(Counter) ' +
    'VALUES (:NewValueParam)';
  CommandType := cmdText;
  Parameters.ParamByName("NewValueParam").Value := 57;
  Execute
end;
ADOCommand1->CommandText = "INSERT INTO Talley ";
ADOCommand1->CommandText += "(Counter) ";
ADOCommand1->CommandText += "VALUES (:NewValueParam)";
ADOCommand1->CommandType = cmdText;
ADOCommand1->Parameters->ParamByName("NewValueParam")->Value = 57;
ADOCommand1->Execute()
```

See Also

Specifying the Command (see page 1489)

Using the Execute Method (see page 1493)

Canceling Commands (see page 1477)

Retrieving Result Sets with Commands (see page 1488)

3.2.3.1.17 Indicating the Types of Operations the Connection Supports

ADO connections are established using a specific mode, similar to the mode you use when opening a file. The connection mode determines the permissions available to the connection, and hence the types of operations (such as reading and writing) that can be performed using that connection.

Use the Mode property to indicate the connection mode. The possible values are listed in the following table:

ADO connection modes

Connect Mode	Meaning
cmUnknown	Permissions are not yet set for the connection or cannot be determined.
cmRead	Read-only permissions are available to the connection.
cmWrite	Write-only permissions are available to the connection.
cmReadWrite	Read/write permissions are available to the connection.
cmShareDenyRead	Prevents others from opening connections with read permissions.
cmShareDenyWrite	Prevents others from opening connection with write permissions.
cmShareExclusive	Prevents others from opening connection.
cmShareDenyNone	Prevents others from opening connection with any permissions.

The possible values for *Mode* correspond to the *ConnectModeEnum* values of the *Mode* property on the underlying ADO connection object. See the Microsoft Data Access SDK help for more information on these values.

See Also

Specifying Whether the Connection Automatically Initiates Transactions (see page 1489)

Controlling Timeouts (see page 1480)

3.2.3.1.18 Inspecting the Update Status of Individual Rows

Determine the update status of a given row by making it current and then inspecting the RecordStatus property of the ADO data component. *RecordStatus* reflects the update status of the current row and only that row.

```
if (rsNew in ADOQuery1.RecordStatus) then
begin
...
end;
else
if (rsDeleted in ADOQuery1.RecordStatus) then
begin
...
else
switch (ADOQuery->RecordStatus)
{
  case rsUnmodified:
    StatusBar1->Panels->Items[0]->Text = "Unchanged record";
    break;
  case rsModified:
    StatusBar1->Panels->Items[0]->Text = "Changed record";
    break;
  case rsDeleted:
    StatusBar1->Panels->Items[0]->Text = "Deleted record";
    break;
  case rsNew:
    StatusBar1->Panels->Items[0]->Text = "New record";
```

```
    break;
}
```

See Also

Opening the Dataset in Batch Update Mode (see page 1487)

Filtering Multiple Rows Based On Update Status (see page 1481)

Applying the Batch Updates to Base Tables (see page 1476)

Canceling Batch Updates (see page 1476)

3.2.3.1.19 Loading Data from and Saving Data to Files

The data retrieved via an ADO dataset component can be saved to a file for later retrieval on the same or a different computer. The data is saved in one of two proprietary formats: ADTG or XML. These two file formats are the only formats supported by ADO. However, both formats are not necessarily supported in all versions of ADO. Consult the ADO documentation for the version you are using to determine what save file formats are supported.

Save the data to a file using the SaveToFile method. *SaveToFile* takes two parameters, the name of the file to which data is saved, and, optionally, the format (ADTG or XML) in which to save the data. Indicate the format for the saved file by setting the Format parameter to *pfADTG* or *pfXML*. If the file specified by the *FileName* parameter already exists, *SaveToFile* raises an *EOleException*.

Retrieve the data from file using the LoadFromFile method. *LoadFromFile* takes a single parameter, the name of the file to load. If the specified file does not exist, *LoadFromFile* raises an *EOleException* exception. On calling the *LoadFromFile* method, the dataset component is automatically activated.

In the example below, the first procedure saves the dataset retrieved by the *TADODataSet* component *ADODataSet1* to a file. The target file is an ADTG file named SaveFile, saved to a local drive. The second procedure loads this saved file into the *TADODataSet* component *ADODataSet2*.

```
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  if (FileExists("c:\SaveFile")) then
  begin
    DeleteFile("c:\SaveFile");
    StatusBar1.Panels[0].Text := "Save file deleted!";
  end;
  ADODataSet1.SaveToFile("c:\SaveFile", pfADTG);
end;
procedure TForm1.LoadBtnClick(Sender: TObject);
begin
  if (FileExists("c:\SaveFile")) then
    ADODataSet2.LoadFromFile("c:\SaveFile")
  else
    StatusBar1.Panels[0].Text := "Save file does not exist!";
end;
void __fastcall TForm1::SaveBtnClick(TObject *Sender)
{
  if (FileExists("c:\\SaveFile"))
  {
    DeleteFile("c:\\SaveFile");
    Statusbar1->Panels->Items[0]->Text = "Save file deleted!";
  }
  ADODataSet1->SaveToFile("c:\\SaveFile");
}
void __fastcall TForm1::LoadBtnClick(TObject *Sender)
{
  if (FileExists("c:\\SaveFile"))
    ADODataSet1->LoadFromFile("c:\\SaveFile");
  else
```

```
    Statusbar1->Panels->Items[0]->Text = "Save file does not exist!";
}
```

The datasets that save and load the data need not be on the same form as above, in the same application, or even on the same computer. This allows for the briefcase-style transfer of data from one computer to another.

See Also

Connecting an ADO Dataset to a Data Store (see page 1479)

Working with Record Sets (see page 1494)

Fetching Records Asynchronously (see page 1481)

Using Batch Updates (see page 1491)

Filtering Records Based On Bookmarks (see page 1482)

3.2.3.1.20 Opening the Dataset in Batch Update Mode

To open an ADO dataset in batch update mode, it must meet these criteria:

1. The component's CursorType property must be *ctKeySet* (the default property value) or *ctStatic*.
2. The LockType property must be *ltBatchOptimistic*.
3. The command must be a SELECT query.

Before activating the dataset component, set the *CursorType* and *LockType* properties as indicated above. Assign a SELECT statement to the component's CommandText property (for *TADODataSet*) or the SQL property (for *TADOQuery*). For *TADOStoredProc* components, set the *ProcedureName* to the name of a stored procedure that returns a result set. These properties can be set at design-time through the **Object Inspector** or programmatically at runtime. The example below shows the preparation of a *TADODataSet* component for batch update mode.

```
with ADODataSet1 do begin
  CursorLocation := clUseClient;
  CursorType := ctStatic;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
ADODataSet1->CursorLocation = clUseClient;
ADODataSet1->CursorType = ctStatic;
ADODataSet1->LockType = ltBatchOptimistic;
ADODataSet1->CommandType = cmdText;
ADODataSet1->CommandText = "SELECT * FROM Employee";
```

After a dataset has been opened in batch update mode, all changes to the data are cached rather than applied directly to the base tables.

See Also

Inspecting the Update Status of Individual Rows (see page 1485)

Filtering Multiple Rows Based On Update Status (see page 1481)

Applying the Batch Updates to Base Tables (see page 1476)

Canceling Batch Updates (see page 1476)

3.2.3.1.21 Overview of ADO Components

The ADO page of the **Tool palette** hosts the *dbGo* components. These components let you connect to an ADO data store, execute commands, and retrieve data from tables in databases using the ADO framework. They require ADO 2.1 (or higher) to

be installed on the host computer. Additionally, client software for the target database system (such as Microsoft SQL Server) must be installed, as well as an OLE DB driver or ODBC driver specific to the particular database system.

Most *dbGo* components have direct counterparts in the components available for other data access mechanisms: a database connection component (*TADOConnection*) and various types of datasets. In addition, *dbGo* includes *TADOCommand*, a simple component that is not a dataset but which represents an SQL command to be executed on the ADO data store.

The following table lists the ADO components.

ADO components

Component	Use
TADOConnection	A database connection component (see page 1506) that establishes a connection with an ADO data store; multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and operate on metadata.
TADODataSet	The primary dataset (see page 1632) for retrieving and operating on data; *TADODataSet* can retrieve data from a single or multiple tables; can connect directly to a data store or use a *TADOConnection* component.
TADOTable	A table-type dataset (see page 1620) for retrieving and operating on a recordset produced by a single database table; *TADOTable* can connect directly to a data store or use a *TADOConnection* component.
TADOQuery	A query-type dataset (see page 1618) for retrieving and operating on a recordset produced by a valid SQL statement; *TADOQuery* can also execute data definition language (DDL) SQL statements. It can connect directly to a data store or use a *TADOConnection* component
TADOStoredProc	A stored procedure-type dataset (see page 1618) for executing stored procedures; *TADOStoredProc* executes stored procedures that may or may not retrieve data. It can connect directly to a data store or use a *TADOConnection* component.
TADOCommand	A simple component for executing commands (SQL statements that do not return result sets); *TADOCommand* can be used with a supporting dataset component, or retrieve a dataset from a table; It can connect directly to a data store or use a *TADOConnection* component.

3.2.3.1.22 Retrieving Result Sets with Commands

Unlike *TADOQuery* components, which use different methods to execute depending on whether they return a result set, *TADOCommand* always uses the Execute command to execute the command, regardless of whether it returns a result set. When the command returns a result set, *Execute* returns an interface to the ADO _RecordSet interface.

The most convenient way to work with this interface is to assign it to the RecordSet property of an ADO dataset.

For example, the following code uses *TADOCommand* (*ADOCommand1*) to execute a SELECT query, which returns a result set. This result set is then assigned to the *RecordSet* property of a *TADODataSet* component (*ADODataSet1*).

```
with ADOCommand1 do begin
  CommandText := 'SELECT Company, State ' +
    'FROM customer ' +
    'WHERE State = :StateParam';
  CommandType := cmdText;
  Parameters.ParamByName('StateParam').Value := 'HI';
  ADODataSet1.Recordset := Execute;
end;
```

```
ADOCommand1->CommandText = "SELECT Company, State ";
ADOCommand1->CommandText += "FROM customer ";
ADOCommand1->CommandText += "WHERE State = :StateParam";
ADOCommand1->CommandType = cmdText;
ADOCommand1->Parameters->ParamByName("StateParam")->Value = "HI";
ADOCommand1->Recordset = ADOCommand1->Execute();
```

As soon as the result set is assigned to the ADO dataset's *Recordset* property, the dataset is automatically activated and the

data is available.

See Also

Specifying the Command (see page 1489)

Using the Execute Method (see page 1493)

Canceling Commands (see page 1477)

Handling Command Parameters (see page 1484)

3.2.3.1.23 Specifying the Command

Specify commands for a *TADOCommand* component using the CommandText property. Like *TADODataSet*, *TADOCommand* lets you specify the command in different ways, depending on the CommandType property. Possible values for *CommandType* include: *cmdText* (used if the command is an SQL statement), *cmdTable* (if it is a table name), and *cmdStoredProc* (if the command is the name of a stored procedure). At design-time, select the appropriate command type from the list in the **Object Inspector**. At runtime, assign a value of type *TCommandType* to the *CommandType* property.

```
with ADOCommand1 do begin
  CommandText := "AddEmployee";
  CommandType := cmdStoredProc;
...
end;
ADOCommand1->CommandText = "AddEmployee";
ADOCommand1->CommandType = cmdStoredProc;
...
```

If no specific type is specified, the server is left to decide as best it can based on the command in *CommandText*.

CommandText can contain the text of an SQL query that includes parameters or the name of a stored procedure that uses parameters. You must then supply parameter values, which are bound to the parameters before executing the command. See Handling command parameters (see page 1484) for details.

See Also

Using the Execute Method (see page 1493)

Canceling Commands (see page 1477)

Retrieving Result Sets with Commands (see page 1488)

Handling Command Parameters (see page 1484)

3.2.3.1.24 Specifying Whether the Connection Automatically Initiates Transactions

Use the Attributes property to control the connection component's use of retaining commits and retaining aborts. When the connection component uses retaining commits, then every time your application commits a transaction, a new transaction is automatically started. When the connection component uses retaining aborts, then every time your application rolls back a transaction, a new transaction is automatically started.

Attributes is a set that can contain one, both, or neither of the constants *xaCommitRetaining* and *xaAbortRetaining*. When *Attributes* contains *xaCommitRetaining*, the connection uses retaining commits. When *Attributes* contains *xaAbortRetaining*, it uses retaining aborts.

Check whether either retaining commits or retaining aborts is enabled using the in operator. Enable retaining commits or aborts by adding the appropriate value to the attributes property; disable them by subtracting the value. The example routines below respectively enable and disable retaining commits in an ADO connection component.

```
procedure TForm1.RetainingCommitsOnButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if not (xaCommitRetaining in Attributes) then
      Attributes := (Attributes + [xaCommitRetaining])
    Open;
  end;
end;
procedure TForm1.RetainingCommitsOffButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if (xaCommitRetaining in Attributes) then
      Attributes := (Attributes - [xaCommitRetaining]);
    Open;
  end;
end;
void __fastcall TForm1::RetainingCommitsOnButtonClick(TObject *Sender)
{
  ADOConnection1->Close()
  if (!ADOConnection1->Attributes.Contains(xaCommitRetaining))
    ADOConnection1->Attributes = TXactAttributes() << xaCommitRetaining;
  ADOConnection1->Open()
}
void __fastcall TForm1::RetainingCommitsOffButtonClick(TObject *Sender)
{
  ADOConnection1->Close()
  if (ADOConnection1->Attributes.Contains(xaCommitRetaining))
    ADOConnection1->Attributes = TXactAttributes() >> xaCommitRetaining;
  ADOConnection1->Open()
}
```

See Also

Controlling Timeouts (see page 1480)

Indicating the Types of Operations the Connection Supports (see page 1485)

Indicating the Types of Operations the Connection Supports (see page 1485)

3.2.3.1.25 Using ADO datasets

ADO dataset components encapsulate the ADO Recordset object. They inherit the common dataset capabilities described in Understanding Datasets (see page 1632), using ADO to provide the implementation. In order to use an ADO dataset, you must familiarize yourself with these common features.

In addition to the common dataset features, all ADO datasets add properties, events, and methods for the following:

- Connecting to an ADO datastore (see page 1479)
- Accessing the underlying Recordset object (see page 1494)
- Filtering records based on bookmarks (see page 1482)
- Fetching records asynchronously (see page 1481)
- Performing batch updates (caching updates) (see page 1491)
- Using files on disk to store data (see page 1486)

There are four ADO datasets:

- *TADOTable*, a table-type dataset (see page 1620) that represents all of the rows and columns of a single database table.
- *TADOQuery*, a query-type dataset (see page 1618) that encapsulates an SQL statement and enables applications to access the resulting records, if any.

- *TADOStoredProc*, a stored procedure-type dataset (see page 1618) that executes a stored procedure defined on a database server.
- *TADODataSet*, a general-purpose dataset that includes the capabilities of the other three types. See Using TADODataSet (see page 1492) for a description of features unique to *TADODataSet*.

  Note: When using ADO to access database information, you do not need to use a dataset such as *TADOQuery* to represent SQL commands that do not return a cursor. Instead, you can use *TADOCommand*, a simple component that is not a dataset. For details on *TADOCommand*, see Using Command Objects (see page 1492).

See Also

Using Command Objects (see page 1492)

Connecting to ADO Data Stores (see page 1477)

3.2.3.1.26 Using Batch Updates

One approach for caching updates is to connect the ADO dataset to a client dataset using a dataset provider. This approach is discussed in Using a client dataset to cache updates (see page 1731).

However, ADO dataset components provide their own support for cached updates, which they call batch updates. The following table lists the correspondences between caching updates using a client dataset and using the batch updates features:

Comparison of ADO and client dataset cached updates

ADO dataset	TClientDataSet	Description
LockType	Not used: client datasets always cache updates	Specifies whether the dataset is opened in batch update mode.
CursorType	Not used: client datasets always work with an in-memory snapshot of data	Specifies how isolated the ADO dataset is from changes on the server.
RecordStatus	UpdateStatus	Indicates what update, if any, has occurred on the current row. *RecordStatus* provides more information than *UpdateStatus*.
FilterGroup	StatusFilter	Specifies which type of records are available. *FilterGroup* provides a wider variety of information.
UpdateBatch	ApplyUpdates	Applies the cached updates back to the database server. Unlike *ApplyUpdates*, *UpdateBatch* lets you limit the types of updates to be applied.
CancelBatch	CancelUpdates	Discards pending updates, reverting to the original values. Unlike *CancelUpdates*, *CancelBatch* lets you limit the types of updates to be canceled.

Using the batch updates features of ADO dataset components is a matter of:

- Opening the dataset in batch update mode (see page 1487)
- Inspecting the update status of individual rows (see page 1485)
- Filtering multiple rows based on update status (see page 1481)
- Applying the batch updates to base tables (see page 1476)
- Canceling batch updates (see page 1476)

See Also

Connecting an ADO Dataset to a Data Store (see page 1479)

Working with Record Sets (see page 1494)

Fetching Records Asynchronously (see page 1481)

Loading Data from and Saving Data to Files (see page 1486)

Filtering Records Based On Bookmarks (see page 1482)

3.2.3.1.27 Using Command Objects

In the ADO environment, commands are textual representations of provider-specific action requests. Typically, they are Data Definition Language (DDL) and Data Manipulation Language (DML) SQL statements. The language used in commands is provider-specific, but usually compliant with the SQL-92 standard for the SQL language.

Although you can always execute commands using *TADOQuery*, you may not want the overhead of using a dataset component, especially if the command does not return a result set. As an alternative, you can use the TADOCommand component, which is a lighter-weight object designed to execute commands, one command at a time. *TADOCommand* is intended primarily for executing those commands that do not return result sets, such as Data Definition Language (DDL) SQL statements. Through an overloaded version of its *Execute* method, however, it is capable of returning a result set that can be assigned to the *RecordSet* property of an ADO dataset component.

In general, working with *TADOCommand* is very similar to working with *TADODataSet*, except that you can't use the standard dataset methods to fetch data, navigate records, edit data, and so on. *TADOCommand* objects connect to a data store in the same way as ADO datasets. See Connecting an ADO dataset to a data store (see page 1479) for details.

The following topics provide details on how to specify and execute commands using *TADOCommand*:

- Specifying the command (see page 1489)
- Using Command objects (see page 1493)
- Canceling commands (see page 1477)
- Retrieving result sets with commands (see page 1488)
- Handling command parameters (see page 1484)

See Also

Using TADODataSet (see page 1492)

Connecting to ADO Data Stores (see page 1477)

Using the Execute Method (see page 1493)

3.2.3.1.28 Using TADODataSet

TADODataSet is a general-purpose dataset (see page 1632) for working with data from an ADO data store. Unlike the other ADO dataset components, *TADODataSet* is not a table-type (see page 1620), query-type (see page 1618), or stored procedure-type (see page 1618) dataset. Instead, it can function as any of these types:

- Like a table-type dataset, *TADODataSet* lets you represent all of the rows and columns of a single database table. To use it in this way, set the CommandType property to *cmdTable* and the CommandText property to the name of the table. *TADODataSet* supports table-type tasks such as
- Assigning indexes (see page 1609) to sort records or form the basis of record-based searches. In addition to the standard index properties and methods, *TADODataSet* lets you sort using temporary indexes by setting the Sort property. Indexed-based searches performed using the Seek method use the current index.
- Emptying the dataset (see page 1592). The DeleteRecordsDeleteRecords method provides greater control than related methods in other table-type datasets, because it lets you specify what records to delete.

The table-type tasks supported by *TADODataSet* are available even when you are not using a *CommandType* of *cmdTable*.

- Like a query-type dataset, *TADODataSet* lets you specify a single SQL command that is executed when you open the

dataset. To use it in this way, set the CommandType property to *cmdText* and the CommandText property to the SQL command you want to execute. At design time, you can double-click on the *CommandText* property in the **Object Inspector** to use the Command Text editor for help in constructing the SQL command. *TADODataSet* supports query-type tasks such as

- Using parameters in the query text (see page 1617).
- Setting up master/detail relationships using parameters (see page 1593).
- Preparing the query in advance to improve performance by setting the Prepared property to *True*.
- Like a stored procedure-type dataset, *TADODataSet* lets you specify a stored procedure that is executed when you open the dataset. To use it in this way, set the CommandType property to *cmdStoredProc* and the CommandText property to the name of the stored procedure. *TADODataSet* supports stored procedure-type tasks such as
- Working with stored procedure parameters (see page 1625).
- Fetching multiple result sets (see page 1597).
- Preparing the stored procedure in advance to improve performance by setting the Prepared property to *True*.

In addition, *TADODataSet* lets you work with data stored in files by setting the *CommandType* property to *cmdFile* and the *CommandText* property to the file name.

Before you set the *CommandText* and *CommandType* properties, you should link the *TADODataSet* to a data store by setting the Connection or ConnectionString property. This process is described in Connecting an ADO dataset to a data store (see page 1479). As an alternative, you can use an RDS DataSpace object to connect the *TADODataSet* to an ADO-based application server. To use an RDS DataSpace object, set the RDSConnection property to a TRDSConnection object.

See Also

Understanding Datasets (see page 1632)

Using TADODataSet

3.2.3.1.29 Using the Execute Method

Before *TADOCommand* can execute its command, it must have a valid connection to a data store. This is established just as with an ADO dataset. See Connecting an ADO dataset to a data store (see page 1479) for details.

To execute the command, call the Execute method. *Execute* is an overloaded method that lets you choose the most appropriate way to execute the command.

For commands that do not require any parameters and for which you do not need to know how many records were affected, call *Execute* without any parameters:

```
with ADOCommand1 do begin
  CommandText := "UpdateInventory";
  CommandType := cmdStoredProc;
  Execute;
end;
ADOCommand1->CommandText = "UpdateInventory";
ADOCommand1->CommandType = cmdStoredProc;
ADOCommand1->Execute();
```

Other versions of *Execute* let you provide parameter values using a Variant array, and to obtain the number of records affected by the command.

For information on executing commands that return a result set, see Retrieving result sets with commands (see page 1488).

See Also

Specifying the Command (see page 1489)

Canceling Commands (see page 1477)

Retrieving Result Sets with Commands (see page 1488)

Handling Command Parameters (see page 1484)

3.2.3.1.30 Working with ADO Components

The *dbGo* components provide data access through the ADO framework. ADO, (Microsoft ActiveX Data Objects) is a set of COM objects that access data through an OLE DB provider. The *dbGo* components encapsulate these ADO objects in the Delphi database architecture.

The ADO layer of an ADO-based application consists of Microsoft ADO 2.1, an OLE DB provider or ODBC driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these must be accessible to the ADO-based application for it to be fully functional.

The ADO objects that figure most prominently are the Connection, Command, and Recordset objects. These ADO objects are wrapped by the *TADOConnection*, *TADOCommand*, and ADO dataset components. The ADO framework includes other "helper" objects, like the Field and Properties objects, but these are typically not used directly in dbGo applications and are not wrapped by dedicated components.

Before reading about the features peculiar to the *dbGo* components, you should familiarize yourself with the common features of database connection components (see page 1506) and datasets (see page 1632).

The following topics describe the unique features of dbGo components and how to work with them:

- Overview of ADO components (see page 1487)
- Connecting to ADO data stores (see page 1477)
- Using TADODataSet (see page 1492)
- Using Command Objects (see page 1492)

See Also

Understanding Datasets (see page 1632)

Connecting to Databases (see page 1506)

Working With ADO Components

Designing Database Applications (see page 1566)

Using Data Controls (see page 1778)

Working with Field Components (see page 1877)

Creating and Using a Client Dataset (see page 1740)

Using Unidirectional Datasets (see page 1823)

Using the Borland Database Engine (see page 1643)

Using Provider Components (see page 1819)

Creating Multi-tiered Applications: Overview (see page 1518)

Using Decision Support Components (see page 1800)

Using XML in Database Applications (see page 1847)

3.2.3.1.31 Working with Record Sets

The Recordset property provides direct access to the ADO recordset object underlying the dataset component. Using this object, it is possible to access properties and call methods of the recordset object from an application. Use of *Recordset* to directly

access the underlying ADO recordset object requires a good working knowledge of ADO objects in general and the ADO recordset object in specific. Using the recordset object directly is not recommended unless you are familiar with recordset object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO recordset objects.

The RecordSetState property indicates the current state of the underlying recordset object. *RecordsetState* corresponds to the *State* property of the ADO recordset object. The value of *RecordsetState* is either *stOpen*, *stExecuting*, or *stFetching*. (*TObjectState*, the type of the *RecordsetState* property, defines other values, but only *stOpen*, *stExecuting,* and *stFetching* pertain to recordsets.) A value of *stOpen* indicates that the recordset is currently idle. A value of stExecuting indicates that it is executing a command. A value of stFetching indicates that it is fetching rows from the associated table (or tables).

Use *RecordsetState* values to perform actions dependent on the current state of the dataset. For example, a routine that updates data might check the RecordsetState property to see whether the dataset is active and not in the process of other activities such as connecting or fetching data.

See Also

Connecting an ADO Dataset to a Data Store (see page 1479)

Filtering Records Based On Bookmarks (see page 1482)

Using Batch Updates (see page 1491)

Loading Data from and Saving Data to Files (see page 1486)

Fetching Records Asynchronously (see page 1481)

3.2.3.2 Connecting to databases

Topics

Name	Description
Controlling Connections (see page 1497)	Before you can establish a connection to a database server, your application must provide certain key pieces of information that describe the desired server. Each type of connection component surfaces a different set of properties to let you identify the server. In general, however, they all provide a way for you to name the server you want and supply a set of connection parameters that control how the connection is formed. Connection parameters vary from server to server. They can include information such as user name and password, the maximum size of BLOB fields, SQL roles, and so on. Once... more (see page 1497)
Disconnecting from a Database Server (see page 1497)	There are two ways to disconnect a server using a connection component: • Set the Connected property to *False*. • Call the Close method. Calling *Close* sets *Connected* to *False*. When *Connected* is set to *False*, the connection component generates a BeforeDisconnect event, where you can perform any cleanup before the connection closes. For example, you can use this event to cache information about all open datasets before they are closed. After the *BeforeConnect* event, the connection component closes all open datasets and disconnects from the server. Finally, the connection component generates an AfterDisconnect event, where you can respond... more (see page 1497)

Managing Transactions (see page 1498)	A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all actions are *rolled back* (undone). By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction. For example, in a banking application, transferring funds from one account to another is an operation you would want to protect with a transaction. If, after decrementing... more (see page 1498)
Obtaining Metadata (see page 1501)	All database connection components can retrieve lists of metadata on the database server, although they vary in the types of metadata they retrieve. The methods that retrieve metadata fill a string list with the names of various entities available on the server. You can then use this information, for example, to let your users dynamically select a table at runtime. You can use a *TADOConnection* component to retrieve metadata about the tables and stored procedures available on the ADO data store. You can then use this information, for example, to let your users dynamically select a table or stored procedure... more (see page 1501)
Sending Commands to the Server (see page 1502)	All database connection components except *TIBDatabase* let you execute SQL statements on the associated server by calling the *Execute* method. Although *Execute* can return a cursor when the statement is a SELECT statement, this use is not recommended. The preferred method for executing statements that return data is to use a dataset. The *Execute* method is very convenient for executing simple SQL statements that do not return any records. Such statements include Data Definition Language (DDL) statements, which operate on or create a database's metadata, such as CREATE INDEX, ALTER TABLE, and DROP DOMAIN. Some Data Manipulation Language (DML) SQL... more (see page 1502)
Specifying the Transaction Isolation Level (see page 1504)	Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transactions' changes to a table. Each server type supports a different set of possible transaction isolation levels. There are three possible transaction isolation levels: • *DirtyRead*: When the isolation level is *DirtyRead*, your transaction sees all changes made by other transactions, even if they have not been committed. Uncommitted changes are not permanent, and might be rolled back at any time. This value provides the least isolation, and is... more (see page 1504)
Using Implicit Connections (see page 1505)	No matter what data access mechanism you are using, you can always create the connection component explicitly and use it to manage the connection to and communication with a database server. For BDE-enabled and ADO-based datasets, you also have the option of describing the database connection through properties of the dataset and letting the dataset generate an implicit connection. For BDE-enabled datasets, you specify an implicit connection using the DatabaseName property. For ADO-based datasets, you use the ConnectionString property. When using an implicit connection, you do not need to explicitly create a connection component. This can simplify your application development,... more (see page 1505)
Working with Associated Datasets (see page 1505)	All database connection components maintain a list of all datasets that use them to connect to a database. A connection component uses this list, for example, to close all of the datasets when it closes the database connection. You can use this list as well, to perform actions on all the datasets that use a specific connection component to connect to a particular database.
Connecting to Databases: Overview (see page 1506)	Most dataset components can connect directly to a database server (see page 1560). Once connected, the dataset communicates with the server automatically. When you open the dataset, it populates itself with data from the server, and when you post records, they are sent back the server and applied. A single connection component can be shared by multiple datasets, or each dataset can use its own connection. Each type of dataset connects to the database server using its own type of connection component, which is designed to work with a single data access mechanism. The following table lists these data access mechanisms and... more (see page 1506)

Controlling Server Login (see page 1507)	Most remote database servers include security features to prohibit unauthorized access. Usually, the server requires a user name and password login before permitting database access. At design time, if a server requires a login, a standard login dialog box prompts for a user name and password when you first attempt to connect to the database. At runtime, there are three ways you can handle a server's request for a login: The first way is to let the default login dialog and processes handle the login. This is the default approach. Set the *LoginPrompt property of the connection component to True... more (see page 1507)*
Connecting to a Database Server (see page 1509)	There are two ways to connect to a database server using a connection component: • Call the Open method. • Set the Connected property to *True*. *Calling the Open method sets Connected to True.* ***Note:*** *When a connection component is not connected to a server and an application attempts to open one of its associated datasets, the dataset automatically calls the connection component's Open method.* *When you set Connected to True, the connection component first generates a BeforeConnect event, where you can perform any initialization. For example, you can use this event to alter connection parameters.* *After the BeforeConnect... more (see page 1509)*

3.2.3.2.1 Controlling Connections

Before you can establish a connection to a database server, your application must provide certain key pieces of information that describe the desired server. Each type of connection component surfaces a different set of properties to let you identify the server. In general, however, they all provide a way for you to name the server you want and supply a set of connection parameters that control how the connection is formed. Connection parameters vary from server to server. They can include information such as user name and password, the maximum size of BLOB fields, SQL roles, and so on.

Once you have identified the desired server and any connection parameters, you can use the connection component to explicitly open or close a connection. The connection component generates events when it opens or closes a connection that you can use to customize the response of your application to changes in the database connection.

The following topics provide details about opening and closing database connections:

- Connecting to a Database Server (see page 1509)
- Disconnecting From a Database Server (see page 1497)

See Also

Using Implicit Connections (see page 1505)

Controlling Server Login (see page 1507)

Managing Transactions (see page 1498)

Working with Associated Datasets (see page 1505)

Sending Commands to the Server (see page 1502)

3.2.3.2.2 Disconnecting from a Database Server

There are two ways to disconnect a server using a connection component:

- Set the Connected property to *False*.
- Call the Close method.

Calling *Close* sets *Connected* to *False*.

When *Connected* is set to *False*, the connection component generates a BeforeDisconnect event, where you can perform any cleanup before the connection closes. For example, you can use this event to cache information about all open datasets before they are closed.

After the *BeforeConnect* event, the connection component closes all open datasets and disconnects from the server.

Finally, the connection component generates an AfterDisconnect event, where you can respond to the change in connection status, such as enabling a Connect button in your user interface.

Note: Calling *Close* or setting *Connected* to *False* disconnects from a database server even if the connection component has a *KeepConnection* property that is *True*.

See Also

Connecting to a Database Server (see page 1509)

3.2.3.2.3 Managing Transactions

A *transaction* is a group of actions that must all be carried out successfully on one or more tables in a database before they are *committed* (made permanent). If one of the actions in the group fails, then all actions are *rolled back* (undone). By using transactions, you ensure that the database is not left in an inconsistent state when a problem occurs completing one of the actions that make up the transaction.

For example, in a banking application, transferring funds from one account to another is an operation you would want to protect with a transaction. If, after decrementing the balance in one account, an error occurred incrementing the balance in the other, you want to roll back the transaction so that the database still reflects the correct total balance.

It is always possible to manage transactions by sending SQL commands directly to the database (see page 1502). Most databases provide their own transaction management model, although some have no transaction support at all. For servers that support it, you may want to code your own transaction management directly, taking advantage of advanced transaction management capabilities on a particular database server, such as schema caching.

If you do not need to use any advanced transaction management capabilities, connection components provide a set of methods and properties you can use to manage transactions without explicitly sending any SQL commands. Using these properties and methods has the advantage that you do not need to customize your application for each type of database server you use, as long as the server supports transactions. (The BDE also provides limited transaction support for local tables with no server transaction support. When not using the BDE, trying to start transactions on a database that does not support them causes connection components to raise an exception.)

Warning: When a dataset provider component applies updates, it implicitly generates transactions for any updates. Be careful that any transactions you explicitly start do not conflict with those generated by the provider.

Starting a transaction

When you start a transaction, all subsequent statements that read from or write to the database occur in the context of that transaction, until the transaction is explicitly terminated or (in the case of overlapping transactions) until another transaction is started. Each statement is considered part of a group. Changes must be successfully committed to the database, or every change made in the group must be undone.

While the transaction is in process, your view of the data in database tables is determined by your transaction isolation level (see page 1504).

For *TADOConnection*, start a transaction by calling the *BeginTrans* method:

```
Level := ADOConnection1.BeginTrans;
Level = ADOConnection1->BeginTrans();
```

BeginTrans returns the level of nesting for the transaction that started. A nested transaction is one that is nested within another, parent, transaction. After the server starts the transaction, the ADO connection receives an *OnBeginTransComplete* event.

For *TDatabase,* use the *StartTransaction* method instead. *TDataBase* does not support nested or overlapped transactions: If you call a *TDatabase* component's *StartTransaction* method while another transaction is underway, it raises an exception. To avoid calling *StartTransaction*, you can check the *InTransaction* property:

```
if not Database1.InTransaction then
  Database1.StartTransaction;
if (!Database1->InTransaction)
  Database1->StartTransaction();
```

TSQLConnection also uses the *StartTransaction* method, but it uses a version that gives you a lot more control. Specifically, *StartTransaction* takes a transaction descriptor, which lets you manage multiple simultaneous transactions and specify the transaction isolation level (see page 1504) on a per-transaction basis. In order to manage multiple simultaneous transactions, set the *TransactionID* field of the transaction descriptor to a unique value. *TransactionID* can be any value you choose, as long as it is unique (does not conflict with any other transaction currently underway). Depending on the server, transactions started by *TSQLConnection* can be nested (as they can be when using ADO) or they can be overlapped.

```
var
  TD: TTransactionDesc;
begin
  TD.TransactionID := 1;
  TD.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(TD);
TTransactionDesc TD;
TD.TransactionID = 1;
TD.IsolationLevel = xilREADCOMMITTED;
SQLConnection1->StartTransaction(TD);
```

By default, with overlapped transactions, the first transaction becomes inactive when the second transaction starts, although you can postpone committing or rolling back the first transaction until later. If you are using *TSQLConnection* with an InterBase database, you can identify each dataset in your application with a particular active transaction, by setting its TransactionLevel property. That is, after starting a second transaction, you can continue to work with both transactions simultaneously, simply by associating a dataset with the transaction you want.

Note: Unlike *TADOConnection*, *TSQLConnection* and *TDatabase* do not receive any events when the transactions starts.

InterBase express offers you even more control than *TSQLConnection* by using a separate transaction component rather than starting transactions using the connection component. You can, however, use *TIBDatabase* to start a default transaction:

```
if not IBDatabase1.DefaultTransaction.InTransaction then
  IBDatabase1.DefaultTransaction.StartTransaction;
if (!IBDatabase1->DefaultTransaction->InTransaction)
  IBDatabase1->DefaultTransaction->StartTransaction();
```

You can have overlapped transactions by using two separate transaction components. Each transaction component has a set of parameters that let you configure the transaction. These let you specify the transaction isolation level, as well as other properties of the transaction.

Ending a transaction

Ideally, a transaction should only last as long as necessary. The longer a transaction is active, the more simultaneous users that access the database, and the more concurrent, simultaneous transactions that start and end during the lifetime of your transaction, the greater the likelihood that your transaction will conflict with another when you attempt to commit any changes.

When the actions that make up the transaction have all succeeded, you can make the database changes permanent by committing the transaction. For *TDatabase*, you commit a transaction using the *Commitmethod:*

```
MyOracleConnection.Commit;
MyOracleConnection->Commit();
```

For *TSQLConnection*, you also use the *Commitmethod, but you must specify which transaction you are committing by supplying*

the transaction descriptor you gave to the StartTransaction method:

```
MyOracleConnection.Commit(TD);
MyOracleConnection->Commit(TD);
```

For *TIBDatabase*, you commit a transaction object using its *Commit* method:

```
IBDatabase1.DefaultTransaction.Commit;
IBDatabase1->DefaultTransaction->Commit();
```

For *TADOConnection*, you commit a transaction using the *CommitTrans* method:

```
ADOConnection1.CommitTrans;
ADOConnection1->CommitTrans();
```

Note: It is possible for a nested transaction to be committed, only to have the changes rolled back later if the parent transaction is rolled back.

After the transaction is successfully committed, an ADO connection component receives an *OnCommitTransComplete* event. Other connection components do not receive any similar events.

A call to commit the current transaction is usually attempted in a **try...except** statement. That way, if the transaction cannot commit successfully, you can use the **except** block to handle the error and retry the operation or to roll back the transaction.

If an error occurs when making the changes that are part of the transaction or when trying to commit the transaction, you will want to discard all changes that make up the transaction. Discarding these changes is called rolling back the transaction.

For *TDatabase*, you roll back a transaction by calling the *Rollback* method:

```
MyOracleConnection.Rollback;
MyOracleConnection->Rollback();
```

For *TSQLConnection*, you also use the *Rollback* method, but you must specify which transaction you are rolling back by supplying the transaction descriptor you gave to the *StartTransaction* method:

```
MyOracleConnection.Rollback(TD);
MyOracleConnection->Rollback(TD);
```

For *TIBDatabase*, you roll back a transaction object by calling its *Rollback* method:

```
IBDatabase1.DefaultTransaction.Rollback;
IBDatabase1->DefaultTransaction->Rollback();
```

For *TADOConnection*, you roll back a transaction by calling the *RollbackTrans* method:

```
ADOConnection1.RollbackTrans;
ADOConnection1->RollbackTrans();
```

After the transaction is successfully rolled back, an ADO connection component receives an *OnRollbackTransComplete* event. Other connection components do not receive any similar events.

A call to roll back the current transaction usually occurs in

- Exception handling code when you can't recover from a database error.
- Button or menu event code, such as when a user clicks a Cancel button.

See Also

Controlling Server Login (see page 1507)

Controlling Connections (see page 1497)

Using Implicit Connections (see page 1505)

Working with Associated Datasets (see page 1505)

Sending Commands to the Server (see page 1502)

Transactions (see page 1570)

Obtaining Metadata (see page 1501)

3.2.3.2.4 Obtaining Metadata

All database connection components can retrieve lists of metadata on the database server, although they vary in the types of metadata they retrieve. The methods that retrieve metadata fill a string list with the names of various entities available on the server. You can then use this information, for example, to let your users dynamically select a table at runtime.

You can use a *TADOConnection* component to retrieve metadata about the tables and stored procedures available on the ADO data store. You can then use this information, for example, to let your users dynamically select a table or stored procedure at runtime.

Listing available tables

The *GetTableNames* method copies a list of table names to an already-existing string list object. This can be used, for example, to fill a list box with table names that the user can then use to choose a table to open. The following line fills a listbox with the names of all tables on the database:

```
MyDBConnection.GetTableNames(ListBox1.Items, False);
MyDBConnection->GetTableNames(ListBox1->Items, false);
```

GetTableNames has two parameters: the string list to fill with table names, and a boolean that indicates whether the list should include system tables, or ordinary tables. Note that not all servers use system tables to store metadata, so asking for system tables may result in an empty list.

Note: For most database connection components, *GetTableNames* returns a list of all available non-system tables when the second parameter is *False*. For *TSQLConnection*, however, you have more control over what type is added to the list when you are not fetching only the names of system tables. When using *TSQLConnection*, the types of names added to the list are controlled by the *TableScope* property. *TableScope* indicates whether the list should contain any or all of the following: ordinary tables, system tables, synonyms, and views.

Listing the fields in a table

The *GetFieldNames* method fills an existing string list with the names of all fields (columns) in a specified table. *GetFieldNames* takes two parameters, the name of the table for which you want to list the fields, and an existing string list to be filled with field names:

```
MyDBConnection.GetFieldNames('Employee', ListBox1.Items);
MyDBConnection->GetTableNames("Employee", ListBox1->Items);
```

Listing available stored procedures

To get a listing of all of the stored procedures contained in the database, use the *GetProcedureNames* method. This method takes a single parameter: an already-existing string list to fill:

```
MyDBConnection.GetProcedureNames(ListBox1.Items);
MyDBConnection->GetProcedureNames(ListBox1->Items);
```

Note: *GetProcedureNames* is only available for *TADOConnection* and *TSQLConnection*.

Listing available indexes

To get a listing of all indexes defined for a specific table, use the *GetIndexNames* method. This method takes two parameters: the table whose indexes you want, and an already-existing string list to fill:

```
SQLConnection1.GetIndexNames('Employee', ListBox1.Items);
MyDBConnection1->GetIndexNames("Employee", ListBox1->Items);
```

Note: *GetIndexNames* is only available for *TSQLConnection*, although most table-type datasets have an equivalent method.

Listing stored procedure parameters

To get a list of all parameters defined for a specific stored procedure, use the *GetProcedureParams* method. *GetProcedureParams* fills a *TList* object with pointers to parameter description records, where each record describes a parameter of a specified stored procedure, including its name, index, parameter type, field type, and so on.

GetProcedureParams takes two parameters: the name of the stored procedure, and an already-existing *TList* object to fill:

```
SQLConnection1.GetProcedureParams('GetInterestRate', List1);
MyDBConnection1->GetIndexNames("GetInterestRate", List1);
```

To convert the parameter descriptions that are added to the list into the more familiar *TParams* object, call the global *LoadParamListItems* procedure. Because *GetProcedureParams* dynamically allocates the individual records, your application must free them when it is finished with the information. The global *FreeProcParams* routine can do this for you.

Note: *GetProcedureParams* is only available for *TSQLConnection*.

See Also

Controlling Server Login (see page 1507)

Controlling Connections (see page 1497)

Using Implicit Connections (see page 1505)

Sending Commands to the Server (see page 1502)

Managing Transactions (see page 1498)

Understanding Datasets (see page 1632)

Working with Associated Datasets (see page 1505)

3.2.3.2.5 Sending Commands to the Server

All database connection components except *TIBDatabase* let you execute SQL statements on the associated server by calling the *Execute* method. Although *Execute* can return a cursor when the statement is a SELECT statement, this use is not recommended. The preferred method for executing statements that return data is to use a dataset.

The *Execute* method is very convenient for executing simple SQL statements that do not return any records. Such statements include Data Definition Language (DDL) statements, which operate on or create a database's metadata, such as CREATE INDEX, ALTER TABLE, and DROP DOMAIN. Some Data Manipulation Language (DML) SQL statements also do not return a result set. The DML statements that perform an action on data but do not return a result set are: INSERT, DELETE, and UPDATE.

The syntax for the *Execute* method varies with the connection type:

- For *TDatabase*, *Execute* takes four parameters: a string that specifies a single SQL statement that you want to execute, a TParams object that supplies any parameter values for that statement, a boolean that indicates whether the statement should be cached because you will call it again, and a pointer to a BDE cursor that can be returned (It is recommended that you pass nil).
- For *TADOConnection*, there are two versions of *Execute.* The first takes a WideString that specifies the SQL statement and a second parameter that specifies a set of options that control whether the statement is executed asynchronously and whether it returns any records. This first syntax returns an interface for the returned records. The second syntax takes a WideString that specifies the SQL statement, a second parameter that returns the number of records affected when the statement executes, and a third that specifies options such as whether the statement executes asynchronously. Note that neither syntax provides for passing parameters.
- For *TSQLConnection*, *Execute* takes three parameters: a string that specifies a single SQL statement that you want to execute, a TParams object that supplies any parameter values for that statement, and a pointer that can receive a *TCustomSQLDataSet* that is created to return records.

Note: *Execute* can only execute one SQL statement at a time. It is not possible to execute multiple SQL statements with a single call to *Execute*, as you can with SQL scripting utilities. To execute more than one statement, call *Execute* repeatedly.

It is relatively easy to execute a statement that does not include any parameters. For example, the following code executes a CREATE TABLE statement (DDL) without any parameters on a *TSQLConnection* component:

```
procedure TForm1.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  SQLConnection1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '( " +
    '  CustNo INTEGER, ' +
    '  Company CHAR(40), ' +
    '  State CHAR(2), ' +
    '  PRIMARY KEY (CustNo) ' +
    ')';
  SQLConnection1.Execute(SQLstmt, nil, nil);
end;
void __fastcall TDataForm::CreateTableButtonClick(TObject *Sender)
{
  SQLConnection1->Connected = true;
  AnsiString SQLstmt = "CREATE TABLE NewCusts " +
    "( " +
    " CustNo INTEGER, " +
    " Company CHAR(40), " +
    " State CHAR(2), " +
    " PRIMARY KEY (CustNo) " +
    ")";
  SQLConnection1->Execute(SQLstmt, NULL, NULL);
}
```

To use parameters, you must create a TParams object. For each parameter value, use the *TParams*.*CreateParam* method to add a *TParam* object. Then use properties of *TParam* to describe the parameter and set its value.

This process is illustrated in the following example, which uses *TDatabase* to execute an INSERT statement. The INSERT statement has a single parameter named: *StateParam*. A *TParams* object (called *stmtParams*) is created to supply a value of "CA" for that parameter.

```
procedure TForm1.INSERT_WithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
begin
  stmtParams := TParams.Create;
  try
    Database1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'INSERT INTO "Custom.db" '+
      '(CustNo, Company, State) ' +
      'VALUES (7777, "Robin Dabank Consulting", :StateParam)';
    Database1.Execute(SQLstmt, stmtParams, False, nil);
  finally
    stmtParams.Free;
  end;
end;
void __fastcall TForm1::INSERT_WithParamsButtonClick(TObject *Sender)
{
  AnsiString SQLstmt;
  TParams *stmtParams = new TParams;
  try
  {
    Database1->Connected = true;
    stmtParams->CreateParam(ftString, "StateParam", ptInput);
```

```
    stmtParams->ParamByName("StateParam")->AsString = "CA";
    SQLstmt = "INSERT INTO "Custom.db" ";
    SQLstmt += "(CustNo, Company, State) ";
    SQLstmt += "VALUES (7777, "Robin Dabank Consulting", :StateParam)";
    Database1->Execute(SQLstmt, stmtParams, false, NULL);
  }
  __finally
  {
    delete stmtParams;
  }
}
```

If the SQL statement includes a parameter but you do not supply a *TParam* object to provide its value, the SQL statement may cause an error when executed (this depends on the particular database back-end used). If a *TParam* object is provided but there is no corresponding parameter in the SQL statement, an exception is raised when the application attempts to use the *TParam*.

See Also

Controlling Server Login (see page 1507)

Controlling Connections (see page 1497)

Using Implicit Connections (see page 1505)

Working with Associated Datasets (see page 1505)

Managing Transactions (see page 1498)

3.2.3.2.6 Specifying the Transaction Isolation Level

Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transactions' changes to a table.

Each server type supports a different set of possible transaction isolation levels. There are three possible transaction isolation levels:

- *DirtyRead*: When the isolation level is *DirtyRead*, your transaction sees all changes made by other transactions, even if they have not been committed. Uncommitted changes are not permanent, and might be rolled back at any time. This value provides the least isolation, and is not available for many database servers (such as Oracle, Sybase, MS-SQL, and InterBase).
- *ReadCommitted*: When the isolation level is *ReadCommitted*, only committed changes made by other transactions are visible. Although this setting protects your transaction from seeing uncommitted changes that may be rolled back, you may still receive an inconsistent view of the database state if another transaction is committed while you are in the process of reading. This level is available for all transactions except local transactions managed by the BDE.
- *RepeatableRead*: When the isolation level is *RepeatableRead*, your transaction is guaranteed to see a consistent state of the database data. Your transaction sees a single snapshot of the data. It cannot see any subsequent changes to data by other simultaneous transactions, even if they are committed. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions. This level is not available on some servers, such as Sybase and MS-SQL and is unavailable on local transactions managed by the BDE.

In addition, *TSQLConnection* lets you specify database-specific custom isolation levels. Custom isolation levels are defined by the *dbExpress* driver. See your driver documentation for details.

Note: For a detailed description of how each isolation level is implemented, see your server documentation.

TDatabase and *TADOConnection* let you specify the transaction isolation level by setting the *TransIsolation* property. When you set *TransIsolation* to a value that is not supported by the database server, you get the next highest level of isolation (if available). If there is no higher level available, the connection component raises an exception when you try to start a transaction.

When using *TSQLConnection*, transaction isolation level is controlled by the *IsolationLevel* field of the transaction descriptor.

When using InterBase express, transaction isolation level is controlled by a transaction parameter.

See Also

Managing Transactions (see page 1498)

3.2.3.2.7 Using Implicit Connections

No matter what data access mechanism you are using, you can always create the connection component explicitly and use it to manage the connection to and communication with a database server. For BDE-enabled and ADO-based datasets, you also have the option of describing the database connection through properties of the dataset and letting the dataset generate an implicit connection. For BDE-enabled datasets, you specify an implicit connection using the DatabaseName property. For ADO-based datasets, you use the ConnectionString property.

When using an implicit connection, you do not need to explicitly create a connection component. This can simplify your application development, and the default connection you specify can cover a wide variety of situations. For complex, mission-critical client/server applications with many users and different requirements for database connections, however, you should create your own connection components to tune each database connection to your application's needs. Explicit connection components give you greater control. For example, you need to access the connection component to perform the following tasks:

- Customize database server login support (see page 1507). (Implicit connections display a default login dialog to prompt the user for a user name and password.)
- Control transactions and specify transaction isolation levels (see page 1498).
- Execute SQL commands on the server without using a dataset (see page 1502).
- Perform actions on all open datasets that are connected to the same database (see page 1505).

In addition, if you have multiple datasets that all use the same server, it can be easier to use an connection component, so that you only have to specify the server to use in one place. That way, if you later change the server, you do not need to update several dataset components: only the connection component.

See Also

Controlling Connections (see page 1497)

Controlling Server Login (see page 1507)

Managing Transactions (see page 1498)

Working with Associated Datasets (see page 1505)

Sending Commands to the Server (see page 1502)

3.2.3.2.8 Working with Associated Datasets

All database connection components maintain a list of all datasets that use them to connect to a database. A connection component uses this list, for example, to close all of the datasets when it closes the database connection.

You can use this list as well, to perform actions on all the datasets that use a specific connection component to connect to a particular database.

Closing all datasets without disconnecting from the server

The connection component automatically closes all datasets when you close its connection. There may be times, however, when you want to close all datasets without disconnecting from the database server.

To close all open datasets without disconnecting from a server, you can use the *CloseDataSets* method.

For *TADOConnection* and *TIBDatabase*, calling *CloseDataSets* always leaves the connection open. For *TDatabase* and

TSQLConnection, you must also set the *KeepConnection* property to *True*.

Iterating through the associated datasets

To perform any actions (other than closing them all) on all the datasets that use a connection component, use the DataSets and DataSetCount properties. *DataSets* is an indexed array of all datasets that are linked to the connection component. For all connection components except *TADOConnection*, this list includes only the active datasets. *TADOConnection* lists the inactive datasets as well. *DataSetCount* is the number of datasets in this array.

Note: When you use a specialized client dataset to cache updates (as opposed to the generic client dataset, *TClientDataSet*), the *DataSets* property lists the internal dataset owned by the client dataset, not the client dataset itself.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets and disables any controls that use the data they provide:

```
var
  I: Integer;
begin
  with MyDBConnection do
  begin
    for I := 0 to DataSetCount - 1 do
      DataSets[I].DisableControls;
  end;
end;
for (int i = 0; i < MyDBConnection->DataSetCount; i++)
  MyDBConnection->DataSets[i]->DisableControls();
```

Note: *TADOConnection* supports command objects as well as datasets. You can iterate through these much like you iterate through the datasets, by using the *Commands* and *CommandCount* properties.

See Also

Controlling Server Login (see page 1507)

Controlling Connections (see page 1497)

Using Implicit Connections (see page 1505)

Sending Commands to the Server (see page 1502)

Managing Transactions (see page 1498)

Understanding Datasets (see page 1632)

Obtaining Metadata (see page 1501)

3.2.3.2.9 Connecting to Databases: Overview

Most dataset components can connect directly to a database server (see page 1560). Once connected, the dataset communicates with the server automatically. When you open the dataset, it populates itself with data from the server, and when you post records, they are sent back the server and applied. A single connection component can be shared by multiple datasets, or each dataset can use its own connection.

Each type of dataset connects to the database server using its own type of connection component, which is designed to work with a single data access mechanism. The following table lists these data access mechanisms and the associated connection components:

Database connection components

Data Access Mechanism	Connection Component
Borland Database Engine (BDE)	TDatabase

ActiveX Data Objects (ADO)	TADOConnection
dbExpress	TSQLConnection
InterBase Express	TIBDatabase

Note: For a discussion of some pros and cons of each of these mechanisms, see Using Databases (see page 1572).

The connection component provides all the information necessary to establish a database connection. This information is different for each type of connection component:

- For information about describing a BDE-based connection, see Identifying the Database (see page 1648).
- For information about describing an ADO-based connection, see Connecting to a Data Store Using TADOConnection (see page 1478).
- For information about describing a dbExpress connection, see Setting up TSQLConnection (see page 1834).
- For information about describing an InterBase Express connection, see TIBDatabase.

Although each type of dataset uses a different connection component, they are all descendants of TCustomConnection. They all perform many of the same tasks and surface many of the same properties, methods, and events.

The following topics discuss many of these common tasks:

- Using Implicit Connections (see page 1505)
- Controlling Connections (see page 1497)
- Controlling Server Login (see page 1507)
- Managing Transactions (see page 1498)
- Working with Associated Datasets (see page 1505)
- Sending Commands to the Server (see page 1502)
- Obtaining Metadata (see page 1501)

See Also

Designing Database Applications (see page 1566)

Understanding Datasets (see page 1632)

Using Data Controls (see page 1778)

Working with Field Components (see page 1877)

Creating and Using a Client Dataset (see page 1740)

Working With ADO Components (see page 1494)

Using Unidirectional Datasets (see page 1823)

Using the Borland Database Engine (see page 1643)

Using Provider Components (see page 1819)

Creating Multi-tiered Applications: Overview (see page 1518)

Using Decision Support Components (see page 1800)

Using XML in Database Applications (see page 1847)

3.2.3.2.10 Controlling Server Login

Most remote database servers include security features to prohibit unauthorized access. Usually, the server requires a user

name and password login before permitting database access.

At design time, if a server requires a login, a standard login dialog box prompts for a user name and password when you first attempt to connect to the database.

At runtime, there are three ways you can handle a server's request for a login:

The first way is to let the default login dialog and processes handle the login. This is the default approach. Set the *LoginPrompt property of the connection component to True (the default) and add DBLogDlg to the uses clause of the unit that declares the connection component. Your application displays the standard login dialog box when the server requests a user name and password.*

The second way is to supply the login information before the login attempt. Each type of connection component uses a different mechanism for specifying the user name and password:

- For BDE, dbExpress, and InterBase express datasets, the user name and password connection parameters can be accessed through the *Params* property. (For BDE datasets, the parameter values can also be associated with a BDE alias, while for dbExpress datasets, they can also be associated with a connection name).
- For ADO datasets, the user name and password can be included in the *C*onnectionString property (or provided as parameters to the *Open* method).

If you specify the user name and password before the server requests them, be sure to set the *LoginPrompt* to *False*, so that the default login dialog does not appear. For example, the following code sets the user name and password on a SQL connection component in the *BeforeConnect* event handler, decrypting an encrypted password that is associated with the current connection name:

```
procedure TForm1.SQLConnectionBeforeConnect(Sender: TObject);
begin
  with Sender as TSQLConnection do
  begin
    if LoginPrompt = False then
    begin
      Params.Values['User_Name'] := 'SYSDBA';
      Params.Values['Password'] := Decrypt(Params.Values['Password']);
    end;
  end;
end;
void __fastcall TForm1::SQLConnectionBeforeConnect(TObject *Sender)
{
  if (SQLConnection1->LoginPrompt == false)
  {
    SQLConnection1->Params->Values["User_Name"] = "SYSDBA";
    SQLConnection1->Params->Values["Password"] =
        Decrypt(SQLConnection1->Params->Values["Password"]);
  }
}
```

Note that setting the user name and password at design-time or using hard-coded strings in code causes the values to be embedded in the application's executable file. This still leaves them easy to find, compromising server security:

The third way is to provide your own custom handling for the login event. The connection component generates an event when it needs the user name and password.

- For *TDatabase*, *TSQLConnection*, and *TIBDatabase*, this is an *OnLogin* event. The event handler has two parameters, the connection component, and a local copy of the user name and password parameters in a string list. (*TSQLConnection* includes the database parameter as well). You must set the LoginPrompt property to *True* for this event to occur. Having a LoginPrompt value of False and assigning a handler for the OnLogin event creates a situation where it is impossible to log in to the database because the default dialog does not appear and the OnLogin event handler never executes.
- For *TADOConnection*, the event is an *On*WillConnect event. The event handler has five parameters, the connection component and four parameters that return values to influence the connection (including two for user name and password). This event always occurs, regardless of the value of *LoginPrompt*.

Write an event handler for the event in which you set the login parameters. Here is an example where the values for the USER

NAME and PASSWORD parameters are provided from a global variable (*UserName*) and a method that returns a password given a user name (*PasswordSearch*)

```
procedure TForm1.Database1Login(Database: TDatabase; LoginParams: TStrings);
begin
  LoginParams.Values['USER NAME'] := UserName;
  LoginParams.Values['PASSWORD'] := PasswordSearch(UserName);
end;
void __fastcall TForm1::Database1Login(TDatabase *Database, TStrings *LoginParams)
{
  LoginParams->Values["USER NAME"] = UserName;
  LoginParams->Values["PASSWORD"] = PasswordSearch(UserName);
}
```

As with the other methods of providing login parameters, when writing an *OnLogin* or *OnWillConnect* event handler, avoid hard coding the password in your application code. It should appear only as an encrypted value, an entry in a secure database your application uses to look up the value, or be dynamically obtained from the user.

See Also

Using Implicit Connections (see page 1505)

Controlling Connections (see page 1497)

Managing Transactions (see page 1498)

Working with Associated Datasets (see page 1505)

Sending Commands to the Server (see page 1502)

3.2.3.2.11 Connecting to a Database Server

There are two ways to connect to a database server using a connection component:

- Call the Open method.
- Set the Connected property to *True*.

Calling the Open method sets Connected to True.

Note: *When a connection component is not connected to a server and an application attempts to open one of its associated datasets, the dataset automatically calls the connection component's Open method.*

When you set Connected to True, the connection component first generates a BeforeConnect event, where you can perform any initialization. For example, you can use this event to alter connection parameters.

After the BeforeConnect event, the connection component may display a default login dialog, depending on how you choose to control server login (see page 1507). It then passes the user name and password to the driver, opening a connection.

Once the connection is open, the connection component generates an AfterConnect event, where you can perform any tasks that require an open connection.

Note: *Some connection components generate additional events as well when establishing a connection.*

Once a connection is established, it is maintained as long as there is at least one active dataset using it. When there are no more active datasets, the connection component drops the connection. Some connection components surface a KeepConnection property that allows the connection to remain open even if all the datasets that use it are closed. If KeepConnection is True, the connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, setting KeepConnection to True reduces network traffic and speeds up the application. If KeepConnection is False, the connection is dropped when there are no active datasets using the database. If a dataset that uses the database is later opened, the connection must be reestablished and initialized.

See Also

Disconnecting from a Database Server (see page 1497)

3.2.3.3 Creating multi-tiered applications

Topics

Name	Description
Connecting to the Application Server (see page 1517)	To establish and maintain a connection to an application server, a client application uses one or more connection components. You can find these components on the DataSnap or WebServices category of the Tool Palette. Use a connection component to • Identify the protocol for communicating with the application server. Each type of connection component represents a different communication protocol. See Choosing a connection protocol (see page 1523) for details on the benefits and limitations of the available protocols. • Indicate how to locate the server machine. The details of identifying the server machine vary depending on the protocol. See the following topics for details: • Specifying (see page 1537)... more (see page 1517)
Creating Multi-tiered Applications: Overview (see page 1518)	A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits (see page 1519), such as centralized business logic and thin client applications. In its simplest form, sometimes called the "three-tiered model," a multi-tiered application is partitioned into thirds: • **Client application:** provides a user interface on the user's machine. • **Application server:** resides in a central networking location accessible to all clients and provides common data services. • **Remote database server:** provides the relational database... more (see page 1518)
Connecting to the Server (see page 1519)	To locate and connect to the application server, you must first set the properties of the connection component to identify the application server. This process is described in Connecting to the application server (see page 1517). Before opening the connection, any client datasets that use the connection component to communicate with the application server should indicate this by setting their *RemoteServer* property to specify the connection component. The connection is opened automatically when client datasets try to access the application server. For example, setting the *Active* property of the client dataset to *True* opens the connection, as long as the *RemoteServer* property... more (see page 1519)
Advantages of the Multi-tiered Database Model (see page 1519)	The multi-tiered database model breaks a database application into logical pieces. The client application can focus on data display and user interactions. Ideally, it knows nothing about how the data is stored or maintained. The application server (middle tier) coordinates and processes requests and updates from multiple clients. It handles all the details of defining datasets and interacting with the database server. The advantages of this multi-tiered model include the following: • **Encapsulation of business logic in a shared middle tier**. Different client applications all access the same middle tier. This allows you to avoid the redundancy (and maintenance cost)... more (see page 1519)

Brokering Connections (see page 1520)	If you have multiple COM-based servers that your client application can choose from, you can use an Object Broker to locate an available server system. The object broker maintains a list of servers from which the connection component can choose. When the connection component needs to connect to an application server, it asks the Object Broker for a computer name (or IP address, host name, or URL). The broker supplies a name, and the connection component forms a connection. If the supplied name does not work (for example, if the server is down), the broker supplies another name, and so... more (see page 1520)
Building a Multi-tiered Application (see page 1520)	
Building an InternetExpress Application (see page 1521)	The following steps describe one way to build a Web application using InternetExpress. The result is an application that creates HTML pages that let users interact with the data from an application server via a javascript-enabled Web browser. You can also build an InternetExpress application using the Site Express architecture by using the InternetExpress page producer (*TInetXPageProducer*).
Building Web Applications Using InternetExpress (see page 1522)	A client application can request that the application server provide data packets that are coded in XML instead of OleVariants. By combining XML-coded data packets, special javascript libraries (see page 1549) of database functions, and the Web server application support, you can create thin client applications that can be accessed using a Web browser that supports javascript. This combination of features is called InternetExpress. Before building an InternetExpress application (see page 1521), you should understand the Web server application architecture and the multi-tiered database architecture. These are described in Creating Internet Server Applications (see page 2251) and Creating multi-tiered Applications (see page 1518) An InternetExpress application extends the basic Web server... more (see page 1522)
Calling Server Interfaces (see page 1522)	Applications do not need to call the *IAppServer* or *IAppServerSOAP* interface directly because the appropriate calls are made automatically when you use the properties and methods of the client dataset. However, while it is not necessary to work directly with the *IAppServer* or *IAppServerSOAP* interface, you may have added your own extensions to the remote data module's interface. When you extend the application server's interface (see page 1531), you need a way to call those extensions using the connection created by your connection component. Unless you are using SOAP, you can do this using the AppServer property of the connection component. *AppServer... more (see page 1522)*
Choosing a Connection Protocol (see page 1523)	Each communications protocol you can use to connect your client applications to the application server provides its own unique benefits. Before choosing a protocol, consider how many clients you expect, how you are deploying your application, and future development plans. The following topics describe the unique features for each connection protocol: • Using DCOM Connections (see page 1546) • Using Socket Connections (see page 1548) • Using Web Connections (see page 1550) • Using SOAP Connections (see page 1548)
Configuring TMTSDataModule (see page 1524)	To add a *TMTSDataModule* component to your application, choose **File** ▶ **New** ▶ **Other** and select Transactional Data Module from the Multitier page of the new items dialog. You will see the Transactional Data Module wizard. You must supply a class name for your remote data module. This is the base name of a descendant of *TMTSDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TMTSDataModule*, which implements *IMyDataServer*, a... more (see page 1524)
Configuring TRemoteDataModule (see page 1525)	To add a *TRemoteDataModule* component to your application, choose **File** ▶ **New** ▶ **Other** and select Remote Data Module from the ActiveX page of the new items dialog. You will see the Remote Data Module wizard. You must supply a class name for your remote data module. This is the base name of a descendant of *TRemoteDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TRemoteDataModule*, which implements *IMyDataServer*, a... more (see page 1525)

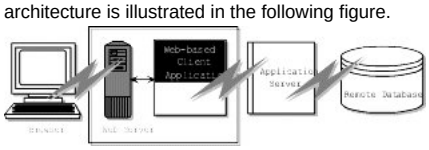
Configuring TSOAPDataModule (see page 1525)	To add a *TSoapDataModule* component to your application, choose **File** ▶ **New** ▶ **Other** and select SOAP Server Data Module from the WebServices page of the new items dialog. The SOAP data module wizard appears. You must supply a class name for your SOAP data module. This is the base name of a *TSoapDataModule* descendant*that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name MyDataServer, the wizard creates a new unit declaring TMyDataServer, a descendant of TSoapDataModule, which implements IMyDataServer, a descendant of... more (see page 1525)*
Connecting to an Application Server That Uses Multiple Data Modules (see page 1526)	If a COM-based application server uses a main "parent" remote data module and several child remote data modules, as described in Using multiple remote data modules (see page 1547), then you need a separate connection component for every remote data module on the application server. Each connection component represents the connection to a single remote data module. While it is possible to have your client application form independent connections to each remote data module on the application server, it is more efficient to use a single connection to the application server that is shared by all the connection components. That is, you... more (see page 1526)
Creating an Active Form for the Client Application (see page 1526)	
Creating the Application Server (see page 1527)	You create an application server very much as you create most database applications. The major difference is that the application server uses a remote data module.
Creating the Client Application (see page 1528)	In most regards, creating a multi-tiered client application is similar to creating a two-tiered client that uses a client dataset to cache updates. The major difference is that a multi-tiered client uses a connection component to establish a conduit to the application server.
Creating Web Pages with an InternetExpress Page Producer (see page 1529)	Each InternetExpress page producer generates an HTML document that appears in the browsers of your application's clients. If your application includes several separate Web documents, use a separate page producer for each of them. The InternetExpress page producer (*TInetXPageProducer*) is a special page producer component (see page 2270). As with other page producers, you can assign it as the Producer property of an action item or call it explicitly from an OnAction event handler. For more information about using content producers with action items, see Responding to request messages with action items (see page 2278). The InternetExpress page producer has a default... more (see page 1529)
Customizing the InternetExpress Page Producer Template (see page 1529)	The template of an InternetExpress page producer is an HTML document with extra embedded tags that your application translates dynamically. Initially, the page producer generates a default template as the value of the HTMLDoc property. This default template has the form
Distributing a Client Application as an ActiveX Control (see page 1530)	The multi-tiered database architecture can be combined with ActiveX features to distribute a client application as an ActiveX control. When you distribute your client application as an ActiveX control, create the application server (see page 1527) as you would for any other multi-tiered application. When creating the client application, you must use an Active Form as the basis instead of an ordinary form. See Creating an Active Form for the Client Application (see page 1526) for details. Once you have built and deployed your client application, it can be accessed from any ActiveX-enabled Web browser on another machine. For a Web browser to successfully launch your... more (see page 1530)
Dropping or Changing a Server Connection (see page 1531)	A connection component drops a connection to the application server when you • set the *Connected* property to *False*. • free the connection component. A connection object is automatically freed when a user closes the client application. • change any of the properties that identify the application server (*ServerName*, *ServerGUID*, *ComputerName*, and so on). Changing these properties allows you to switch among available application servers at runtime. The connection component drops the current connection and establishes a new one. **Note:** Instead of using a single connection component to switch among available application servers, a client application can instead... more (see page 1531)

Extending the Interface of the Application Server (see page 1531)	Client applications interact with the application server by creating or connecting to an instance of the remote data module. They use its interface as the basis of all communication with the application server. You can add to your remote data module's interface to provide additional support for your client applications. This interface is a descendant of *IAppServer* and is created for you automatically by the wizard when you create the remote data module (see page 1536). To add to the remote data module's interface, you can • Choose the Add to Interface command from the Edit menu in the IDE. Indicate whether you... more (see page 1531)
Granting Permission to Access and Launch the Application Server (see page 1532)	Requests from the InternetExpress application appear to the application server as originating from a guest account with the name IUSR_computername, where computername is the name of the system running the Web application. By default, this account does not have access or launch permission for the application server. If you try to use the Web application without granting these permissions, when the Web browser tries to load the requested page it times out with EOLE_ACCESS_ERROR. **Note:** Because the application server runs under this guest account, it can't be shut down by other accounts. To grant the Web application access and launch... more (see page 1532)
Managing Server Connections (see page 1533)	The main purpose of connection components is to locate and connect to the application server. Because they manage server connections, you can also use connection components to call the methods of the application server's interface. The following topics describe how to use a connection component for • Connecting to the Server (see page 1519). • Dropping or Changing a Server Connection (see page 1531). • Calling Server Interfaces (see page 1522). • Connecting to an Application Server that Uses Multiple Data Modules (see page 1526).
Managing Transactions in Multi-tiered Applications (see page 1533)	When client applications apply updates to the application server, the provider component automatically wraps the process of applying updates and resolving errors in a transaction. This transaction is committed if the number of problem records does not exceed the *MaxErrors* value specified as an argument to the *ApplyUpdates* method. Otherwise, it is rolled back. In addition, you can add transaction support to your server application by adding a database connection component or managing the transaction directly by sending SQL to the database server. This works the same way that you would manage transactions in a two-tiered application. For more information... more (see page 1533)
Overview of a Three-tiered Application (see page 1534)	The following numbered steps illustrate a normal sequence of events for a provider-based three-tiered application: 1. A user starts the client application. The client connects to the application server (which can be specified at design time or runtime). If the application server is not already running, it starts. The client receives an *IAppServer* interface for communicating with the application server. 2. The client requests data from the application server. A client may request all data at once, or may request chunks of data throughout the session (fetch on demand). 3. The application server retrieves the data (first establishing a database connection, if necessary),... more (see page 1534)

Pooling Remote Data Modules (see page 1535)	Object pooling allows you to create a cache of remote data modules that are shared by their clients, thereby conserving resources. How this works depends on the type of remote data module and on the connection protocol (see page 1523). If you are creating a transactional data module (see page 1546) that will be installed to COM+, you can use the COM+ Component Manager to install the application server as a pooled object. Even if you are not using a transactional data module, you can take advantage of object pooling if the connection is formed using *TWebConnection*. Under this second type of object pooling,... more (see page 1535)
Registering the Application Server (see page 1535)	Before client applications can locate and use an application server, it must be registered or installed. • If the application server uses DCOM, HTTP, or sockets as a communication protocol, it acts as an Automation server and must be registered like any other COM server. For information about registering a COM server, see Registering a COM Object (see page 1440). • If you are using a transactional data module, you do not register the application server. Instead, you install it with COM+ or MTS. . • When the application server uses SOAP, the application must be a Web Service application. As such, it must be... more (see page 1535)
Setting Up the Remote Data Module (see page 1536)	When you create the remote data module, you must provide certain information that indicates how it responds to client requests. This information varies, depending on the type of remote data module. See The Structure of the Application Server (see page 1542) for information on what type of remote data module you need. The following topics describe how to configure each type of remote data module: • Configuring TRemoteDataModule (see page 1525) • Configuring TMTSDataModule (see page 1524) • Configuring TSoapDataModule (see page 1525)
Setting Web Item Properties (see page 1536)	The Web items that you add using the Web page editor are specialized components that generate HTML. Each Web item class is designed to produce a specific control or section of the final HTML document, but a common set of properties influences the appearance of the final HTML. When a Web item represents information from the XML data packet (for example, when it generates a set of field or parameter display controls or a button that manipulates the data), the XMLBroker property associates the Web item with the XML broker that manages the data packet. You can further specify a... more (see page 1536)
Specifying a Connection Using HTTP (see page 1537)	You can establish a connection to the application server using HTTP from any machine that has a TCP/IP address. Unlike sockets, however, HTTP allows you to take advantage of SSL security and to communicate with a server that is protected behind a firewall. When using HTTP, include a TWebConnection component for connecting to the application server. The Web connection component establishes a connection to the Web server application (httpsrvr.dll), which in turn communicates with the application server. *TWebConnection* locates httpsrvr.dll using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https),... more (see page 1537)
Specifying a Connection Using DCOM (see page 1537)	When using DCOM to communicate with the application server, client applications include a TDCOMConnection component for connecting to the application server. *TDCOMConnection* uses the ComputerName property to identify the machine on which the server resides. When *ComputerName* is blank, the DCOM connection component assumes that the application server resides on the client machine or that the application server has a system registry entry. If you do not provide a system registry entry for the application server on the client when using DCOM, and the server resides on a different machine from the client, you must supply *ComputerName*. **Note:** Even... more (see page 1537)

Specifying a Connection Using SOAP (see page 1538)	You can establish a connection to a SOAP application server using the TSoapConnection component. *TSoapConnection* is very similar to *TWebConnection*, because it also uses HTTP as a transport protocol. Thus, you can use *TSoapConnection* from any machine that has a TCP/IP address, and it can take advantage of SSL security and to communicate with a server that is protected by a firewall. The SOAP connection component establishes a connection to a Web Service provider that implements the *IAppServerSOAP* or *IAppServer* interface. (The UseSOAPAdapter property specifies which interface it expects the server to support.) If the server implements the *IAppServerSOAP... more (see page 1538)*
Specifying a Connection Using Sockets (see page 1538)	You can establish a connection to the application server using sockets from any machine that has a TCP/IP address. This method has the advantage of being applicable to more machines, but does not provide for using any security protocols. When using sockets, include a TSocketConnection component for connecting to the application server. *TSocketConnection* identifies the server machine using the IP Address or host name of the server system, and the port number of the socket dispatcher program (Scktsrvr.exe) that is running on the server machine. For more information about IP addresses and port values, see Describing sockets (see page 2336). Three properties... more (see page 1538)
Supporting Master/detail Relationships (see page 1539)	You can create master/detail relationships between client datasets in your client application in the same way you set them up using any table-type dataset. For more information about setting up master/detail relationships in this way, see Creating Master/detail Relationships (see page 1590). However, this approach has two major drawbacks: • The detail table must fetch and store all of its records from the application server even though it only uses one detail set at a time. (This problem can be mitigated by using parameters. For more information, see Limiting records with parameters (see page 1720).) • It is very difficult to apply updates, because client datasets... more (see page 1539)
Supporting State Information in Remote Data Modules (see page 1540)	The *IAppServer* interface, which client datasets use to communicate with providers on the application server, is mostly stateless. When an application is stateless, it does not "remember" anything that happened in previous calls by the client. This stateless quality is useful if you are pooling database connections in a transactional data module, because your application server does not need to distinguish between database connections for persistent information such as record currency. Similarly, this stateless quality is important when you are sharing remote data module instances between many clients, as occurs with just-in-time activation or object pooling. SOAP data modules must... more (see page 1540)
The Structure of the Application Server (see page 1542)	When you set up and run an application server, it does not establish any connection with client applications. Rather, client applications initiate and maintain the connection. The client application uses a connection component to connect to the application server, and uses the interface of the application server to communicate with a selected provider. All of this happens automatically, without your having to write code to manage incoming requests or supply interfaces. The basis of an application server is a remote data module, which is a specialized data module that supports the *IAppServer* interface (for application servers that also function as... more (see page 1542)
The Structure of the Client Application (see page 1543)	To the end user, the client application of a multi-tiered application looks and behaves no differently than a two-tiered application that uses cached updates. User interaction takes place through standard data-aware controls that display data from a TClientDataSet component. For detailed information about using the properties, events, and methods of client datasets, see Using Client Datasets (see page 1740). TClientDataSet fetches data from and applies updates to a provider component, just as in two-tiered applications that use a client dataset with an external provider. For details about providers, see Using Provider Components (see page 1819). For details about client dataset features that facilitate its... more (see page 1543)

Understanding Multi-tiered Database Applications (see page 1544)	Multi-tiered applications use the components on the DataSnap page, the Data Access page, and possibly the WebServices page of the **Tool palette**, plus a remote data module that is created by a wizard on the Multitier or WebServices page of the New Items dialog. They are based on the ability of provider components to package data into transportable data packets and handle updates received as transportable delta packets. The components needed for a multi-tiered application are described in the following table: ***Components used in multi-tiered applications***
Using an XML Broker (see page 1544)	The XML broker serves two major functions: • It fetches XML data packets from the application server and makes them available to the Web Items that generate HTML for the InternetExpress application. • It receives updates in the form of XML delta packets from browsers and applies them to the application server.
Using DCOM Connections (see page 1546)	DCOM provides the most direct approach to communication, requiring no additional runtime applications on the server. DCOM provides the only approach that lets you use security services when writing a transactional data module. These security services are based on assigning roles to the callers of transactional objects. When using DCOM, DCOM identifies the caller to the system that calls your application server (COM+ or MTS). Therefore, it is possible to accurately determine the role of the caller. When using other protocols, however, there is a runtime executable, separate from the application server, that receives client calls. This runtime executable makes... more (see page 1546)
Using Transactional Data Modules (see page 1546)	You can write an application server that takes advantage of special services for distributed applications that are supplied by COM+ (under Windows 2000 and later) or MTS (before Windows 2000). To do so, create a transactional data module instead of an ordinary remote data module. When you use a transactional data module, your application can take advantage of the following special services: • **Security.** COM+ (or MTS) provides role-based security for your application server. Clients are assigned roles, which determine how they can access the MTS data module's interface. The MTS data module implements the IsCallerInRole method, which you lets you... more (see page 1546)
Using Multiple Remote Data Modules (see page 1547)	You may want to structure your application server so that it uses multiple remote data modules. Using multiple remote data modules lets you partition your code, organizing a large application server into multiple units, where each unit is relatively self-contained. Although you can always create multiple remote data modules on the application server that function independently, a special connection component on the DataSnap category of the **Tool palette** provides support for a model where you have one main "parent" remote data module that dispatches connections from clients to other "child" remote data modules. This model requires that you use a... more (see page 1547)
Using SOAP Connections (see page 1548)	SOAP is the protocol that underlies the built-in support for Web Service applications. SOAP marshals method calls using an XML encoding. SOAP connections use HTTP as a transport protocol. SOAP connections have the advantage that they work in cross-platform applications because they are supported on both the Windows and Linux. Because SOAP connections use HTTP, they have the same advantages as Web connections: HTTP provides a lowest common denominator that you know is available on all clients, and clients can communicate with an application server that is protected by a "firewall." For more information about using SOAP to distribute applications,... more (see page 1548)
Using Socket Connections (see page 1548)	TCP/IP Sockets let you create lightweight clients. For example, if you are writing a Web-based client application (see page 1551), you can't be sure that client systems support DCOM. Sockets provide a lowest common denominator that you know will be available for connecting to the application server. For more information about sockets, see Working with Sockets. (see page 2337) Instead of instantiating the remote data module directly from the client (as happens with DCOM), sockets use a separate application on the server (ScktSrvr.exe), which accepts client requests and instantiates the remote data module using COM. The connection component on the client and ScktSrvr.exe... more (see page 1548)

Using the Javascript Libraries (see page 1549)	The HTML pages generated by the InternetExpress components and the Web items they contain make use of several javascript libraries that ship in the source/webmidas directory: ***Javascript libraries***
Using the Web Page Editor (see page 1549)	The Web page editor lets you add Web items to your InternetExpress page producer and view the resulting HTML page. Display the Web page editor by double-clicking on a InternetExpress page producer component. **Note:** You must have Internet Explorer 4 or better installed to use the Web page editor. The top of the Web page editor displays the Web items that generate the HTML document. These Web items are nested, where each type of Web item assembles the HTML generated by its subitems. Different types of items can contain different subitems. On the left, a tree view displays all of... more (see page 1549)
Using Web Connections (see page 1550)	HTTP lets you create clients that can communicate with an application server that is protected by a firewall. HTTP messages provide controlled access to internal applications so that you can distribute your client applications safely and widely. Like socket connections, HTTP messages provide a lowest common denominator that you know will be available for connecting to the application server. For more information about HTTP messages, see Creating Internet Server Applications (see page 2251) Instead of instantiating the remote data module directly from the client (as happens with DCOM), HTTP-based connections use a Web server application on the server (httpsrvr.dll) that accepts client requests... more (see page 1550)
Writing Web-based Client Applications (see page 1551)	If you want to create Web-based clients for your multi-tiered database application, you must replace the client tier with a special Web application that acts simultaneously as a client to an application server and as a Web server application that is installed with a Web server on the same machine. This architecture is illustrated in the following figure.  There are two approaches that you can take to build the Web application: • You can combine the multi-tiered database architecture with an ActiveX form to distribute the client application as an ActiveX control (see page 1530). This allows any browser that supports ActiveX to... more (see page 1551)

3.2.3.3.1 Connecting to the Application Server

To establish and maintain a connection to an application server, a client application uses one or more connection components. You can find these components on the DataSnap or WebServices category of the Tool Palette.

Use a connection component to

- Identify the protocol for communicating with the application server. Each type of connection component represents a different communication protocol. See Choosing a connection protocol (see page 1523) for details on the benefits and limitations of the available protocols.
- Indicate how to locate the server machine. The details of identifying the server machine vary depending on the protocol. See the following topics for details:
- Specifying a connection using DCOM (see page 1537)
- Specifying a connection using sockets (see page 1538)
- Specifying a connection using HTTP (see page 1537)
- Specifying a connection using SOAP (see page 1538)
- Identify the application server on the server machine.
- If you are not using SOAP, identify the server using the ServerName or ServerGUID property. *ServerName* identifies the base name of the class you specify when creating the remote data module on the application server. See Setting up the remote

data module (see page 1536) for details on how this value is specified on the server. If the server is registered or installed on the client machine, or if the connection component is connected to the server machine, you can set the *ServerName* property at design time by choosing from a drop-down list in the **Object Inspector**. *ServerGUID* specifies the GUID of the remote data module's interface. You can look up this value using the type library editor.

- Manage server connections (see page 1533). Connection components can be used to create or drop connections and to call application server interfaces.

If you are using SOAP, the server is identified in the URL you use to locate the server machine. Follow the steps in Specifying a connection using SOAP (see page 1538).

Usually the application server is on a different machine from the client application, but even if the server resides on the same machine as the client application (for example, during the building and testing of the entire multi-tier application), you can still use the connection component to identify the application server by name, specify a server machine, and use the application server interface.

See Also

The Structure of the Client Application (see page 1543)

Managing Server Connections (see page 1533)

Using a Client Dataset with a Data Provider (see page 1732)

Calling Server Interfaces (see page 1522)

3.2.3.3.2 Creating Multi-tiered Applications: Overview

A multi-tiered client/server application is partitioned into logical units, called tiers, which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits (see page 1519), such as centralized business logic and thin client applications.

In its simplest form, sometimes called the "three-tiered model," a multi-tiered application is partitioned into thirds:

- **Client application:** provides a user interface on the user's machine.
- **Application server:** resides in a central networking location accessible to all clients and provides common data services.
- **Remote database server:** provides the relational database management system (RDBMS).

In this three-tiered model, the application server manages the flow of data between clients and the remote database server, so it is sometimes called a "data broker." You usually only create the application server and its clients, although, if you are really ambitious, you could create your own database back end as well.

In more complex multi-tiered applications, additional services reside between a client and a remote database server. For example, there might be a security services broker to handle secure Internet transactions, or bridge services to handle sharing of data with databases on other platforms.

Support for developing multi-tiered applications is an extension of the way client datasets communicate with a provider component using transportable data packets. See Understanding multi-tiered database applications (see page 1544) for an overview of this technology and the architecture of a typical three-tiered application. Once you understand how to create and manage a three-tiered application, you can create and add additional service layers based on your needs.

Building a multi-tiered application (see page 1520) provides details on how to apply this architecture to build a three-tiered application. Writing Web-based client applications (see page 1551) describes how to combine this architecture with other technologies to create a Web-based multi-tiered application.

See Also

Designing Database Applications (see page 1566)

Understanding Datasets (see page 1632)

Using Data Controls (see page 1778)

Working with Field Components (see page 1877)

Creating and Using a Client Dataset (see page 1740)

Connecting to Databases (see page 1506)

Working With ADO Components (see page 1494)

Using Unidirectional Datasets (see page 1823)

Using the Borland Database Engine (see page 1643)

Using Provider Components (see page 1819)

Using XML in Database Applications (see page 1847)

3.2.3.3.3 Connecting to the Server

To locate and connect to the application server, you must first set the properties of the connection component to identify the application server. This process is described in Connecting to the application server (see page 1517). Before opening the connection, any client datasets that use the connection component to communicate with the application server should indicate this by setting their *RemoteServer* property to specify the connection component.

The connection is opened automatically when client datasets try to access the application server. For example, setting the *Active* property of the client dataset to *True* opens the connection, as long as the *RemoteServer* property has been set.

If you do not link any client datasets to the connection component, you can open the connection by setting the *Connected* property of the connection component to *True*.

Before a connection component establishes a connection to an application server, it generates a *BeforeConnect* event. You can perform any special actions prior to connecting in a *BeforeConnect* handler that you code. After establishing a connection, the connection component generates an *AfterConnect* event for any special actions.

See Also

Dropping or Changing a Server Connection (see page 1531)

Calling Server Interfaces (see page 1522)

Connecting to an Application Server That Uses Multiple Data Modules (see page 1526)

3.2.3.3.4 Advantages of the Multi-tiered Database Model

The multi-tiered database model breaks a database application into logical pieces. The client application can focus on data display and user interactions. Ideally, it knows nothing about how the data is stored or maintained. The application server (middle tier) coordinates and processes requests and updates from multiple clients. It handles all the details of defining datasets and interacting with the database server.

The advantages of this multi-tiered model include the following:

- **Encapsulation of business logic in a shared middle tier**. Different client applications all access the same middle tier. This allows you to avoid the redundancy (and maintenance cost) of duplicating your business rules for each separate client application.
- **Thin client applications**. Your client applications can be written to make a small footprint by delegating more of the processing to middle tiers. Not only are client applications smaller, but they are easier to deploy because they don't need to worry about installing, configuring, and maintaining the database connectivity software (such as the database server's client-side software). Thin client applications can be distributed over the Internet for additional flexibility.
- **Distributed data processing**. Distributing the work of an application over several machines can improve performance because of load balancing, and allow redundant systems to take over when a server goes down.
- **Increased opportunity for security**. You can isolate sensitive functionality into tiers that have different access restrictions.

This provides flexible and configurable levels of security. Middle tiers can limit the entry points to sensitive material, allowing you to control access more easily. If you are using HTTP or COM+, you can take advantage of the security models they support.

See Also

Understanding MIDAS Technology (see page 1544)

Building a Multi-tiered Application (see page 1520)

Writing Web-based Client Applications (see page 1551)

3.2.3.3.5 Brokering Connections

If you have multiple COM-based servers that your client application can choose from, you can use an Object Broker to locate an available server system. The object broker maintains a list of servers from which the connection component can choose. When the connection component needs to connect to an application server, it asks the Object Broker for a computer name (or IP address, host name, or URL). The broker supplies a name, and the connection component forms a connection. If the supplied name does not work (for example, if the server is down), the broker supplies another name, and so on, until a connection is formed.

Once the connection component has formed a connection with a name supplied by the broker, it saves that name as the value of the appropriate property (*ComputerName*, *Address*, *Host,RemoteHost*, or *URL*). If the connection component closes the connection later, and then needs to reopen the connection, it tries using this property value, and only requests a new name from the broker if the connection fails.

Use an Object Broker by specifying the ObjectBroker property of your connection component. When the *ObjectBroker* property is set, the connection component does not save the value of *ComputerName*, *Address*, *Host*, *RemoteHost*, or *URL* to disk.

Note: You can not use the *ObjectBroker* property with SOAP connections.

See Also

Specifying a Connection Using DCOM (see page 1537)

Specifying a Connection Using Sockets (see page 1538)

3.2.3.3.6 Building a Multi-tiered Application

To create a multi-tiered database application

1. Create the application server (see page 1527).
2. Register or install the application server (see page 1535).
3. Create a client application (see page 1528).

The order of creation is important. You should create and run the application server before you create a client. At design time, you can then connect to the application server to test your client. You can, of course, create a client without specifying the application server at design time, and only supply the server name at runtime. However, doing so prevents you from seeing if your application works as expected when you code at design time, and you will not be able to choose servers and providers using the **Object Inspector**.

Note: If you are not creating the client application on the same system as the server, and you are using a DCOM connection, you may want to register the application server on the client system. This makes the connection component aware of the application server at design time so that you can choose server names and provider names from a drop-down list in the Object Inspector

. (If you are using a Web connection, SOAP connection, or socket connection, the connection component fetches the names of registered providers from the server machine.)

See Also

Understanding MIDAS Technology (see page 1544)

Advantages of the Multi-tiered Database Model (see page 1519)

Writing Web-based client applications (see page 1551)

3.2.3.3.7 Building an InternetExpress Application

The following steps describe one way to build a Web application using InternetExpress. The result is an application that creates HTML pages that let users interact with the data from an application server via a javascript-enabled Web browser. You can also build an InternetExpress application using the Site Express architecture by using the InternetExpress page producer (*TInetXPageProducer*).

To build a Web application using InternetExpress

1. Choose **File** ▶ **New** ▶ **Other** to display the New Items dialog box, and on the New page select Web Server application. This process is described in Creating Web server applications with Web Broker.
2. From the DataSnap category of the **Tool palette**, add a connection component to the Web Module that appears when you create a new Web server application. The type of connection component you add depends on the communication protocol you want to use. See Choosing a connection protocol (see page 1523) for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see Connecting to the application server (see page 1517).
4. Instead of a client dataset, add an TXMLBroker from the InternetExpress category of the **Tool palette** to the Web module. Like *TClientDataSet*, *TXMLBroker* represents the data from a provider on the application server and interacts with the application server through an *IAppServer* interface. However, unlike client datasets, XML brokers request data packets as XML instead of as OleVariants and interact with InternetExpress components instead of data controls.
5. Set the *RemoteServer* property of the XML broker to point to the connection component you added in step 2. Set the *ProviderName* property to indicate the provider on the application server that provides data and applies updates. For more information about setting up the XML broker, see Using an XML broker (see page 1544).
6. Add an InternetExpress page producer (*TInetXPageProducer*) to the Web module for each separate page that users will see in their browsers. For each page producer, you must set the *IncludePathURL* property to indicate where it can find the javascript libraries (see page 1549) that augment its generated HTML controls with data management capabilities.
7. Right-click a Web page and choose Action Editor to display the Action editor. Add action items for every message you want to handle from browsers. Associate the page producers you added in step 6 with these actions by setting their *Producer* property or writing code in an *OnAction* event handler. For more information on adding action items using the Action editor, see Adding actions to the dispatcher (see page 2277).
8. Double-click each Web page to display the Web Page editor. (You can also display this editor by clicking the ellipsis button in the **Object Inspector** next to the *WebPageItems* property.) In this editor you can add Web Items to design the pages that users see in their browsers. For more information about designing Web pages for your InternetExpress application, see Creating Web pages with an InternetExpress page producer (see page 1529).
9. Build your Web application. Once you install this application with your Web server, browsers can call it by specifying the name of the application as the script name portion of the URL (see page 2253) and the name of the Web Page component as the pathinfo portion.

See Also

Granting Permission to Access and Launch the Application Server (see page 1532)

Creating the Application Server (see page 1527)

Using the Javascript Libraries (see page 1549)

3.2.3.3.8 Building Web Applications Using InternetExpress

A client application can request that the application server provide data packets that are coded in XML instead of OleVariants. By combining XML-coded data packets, special javascript libraries (see page 1549) of database functions, and the Web server application support, you can create thin client applications that can be accessed using a Web browser that supports javascript. This combination of features is called InternetExpress.

Before building an InternetExpress application (see page 1521), you should understand the Web server application architecture and the multi-tiered database architecture. These are described in Creating Internet Server Applications (see page 2251) and Creating multi-tiered Applications (see page 1518)

An InternetExpress application extends the basic Web server application architecture to act as the client of an application server. InternetExpress applications generate HTML pages that contain a mixture of HTML, XML, and javascript. The HTML governs the layout and appearance of the pages seen by end users in their browsers. The XML encodes the data packets and delta packets that represent database information. The javascript allows the HTML controls to interpret and manipulate the data in these XML data packets on the client machine.

If the InternetExpress application uses DCOM to connect to the application server, you must take additional steps to ensure that the application server grants access and launch permissions (see page 1532) to its clients.

Tip: You can create an InternetExpress application to provide Web browsers with "live" data even if you do not have an application server. Simply add the provider and its dataset to the Web module.

See Also

Building a Multi-tiered Application (see page 1520)

Distributing a Client Application as an ActiveX Control (see page 1530)

3.2.3.3.9 Calling Server Interfaces

Applications do not need to call the *IAppServer* or *IAppServerSOAP* interface directly because the appropriate calls are made automatically when you use the properties and methods of the client dataset. However, while it is not necessary to work directly with the *IAppServer* or *IAppServerSOAP* interface, you may have added your own extensions to the remote data module's interface. When you extend the application server's interface (see page 1531), you need a way to call those extensions using the connection created by your connection component. Unless you are using SOAP, you can do this using the AppServer property of the connection component.

AppServer is a Variant that represents the application server's interface. If you are not using SOAP, you can call an interface method using *AppServer* by writing a statement such as

```
MyConnection.AppServer.SpecialMethod(x,y);
```

However, this technique provides late (dynamic) binding of the interface call. That is, the *SpecialMethod* procedure call is not bound until runtime when the call is executed. Late binding is very flexible, but by using it you lose many benefits such as code insight and type checking. In addition, late binding is slower than early binding, because the compiler generates additional calls to the server to set up interface calls before they are invoked.

Using early binding with DCOM

When you are using DCOM as a communications protocol, you can use early binding of *AppServer* calls. Use the as operator to cast the *AppServer* variable to the *IAppServer* descendant you created when you created the remote data module (see page 1536). For example:

```
with MyConnection.AppServer as IMyAppServer do
SpecialMethod(x,y);
```

To use early binding under DCOM, the server's type library must be registered on the client machine. You can use TRegsvr.exe,

which ships with Delphi to register the type library.

Note: See the TRegSvr demo (which provides the source for TRegsvr.exe) for an example of how to register the type library programmatically.

Using dispatch interfaces with TCP/IP or HTTP

When you are using TCP/IP or HTTP, you can't use true early binding, but because the remote data module uses a dual interface, you can use the application server's dispinterface to improve performance over simple late binding. The dispinterface has the same name as the remote data module's interface, with the string 'Disp' appended. You can assign the *AppServer* property to a variable of this type to obtain the dispinterface. Thus:

```
var
TempInterface: IMyAppServerDisp;
begin
  TempInterface :=IMyAppServerDisp(IDispatch(MyConnection.AppServer));
...
TempInterface.SpecialMethod(x,y);
...
end;
```

Note: To use the dispinterface, you must add the _TLB unit that is generated when you save the type library to the **uses** clause of your client module.

Calling the interface of a SOAP-based server

If you are using SOAP, you can't use the *AppServer* property. Instead, you must obtain the server's interface by calling the *GetSOAPServer* method. Before you call *GetSOAPServer*, however, you must take the following steps:

- Your client application must include the definition of the application server's interface and register it with the invocation registry. You can add the definition of this interface to your client application by referencing a WSDL document that describes the interface you want to call. For information on importing a WSDL document that describes the server interface, see Importing WSDL documents (see page 2307). When you import the interface definition, the WSDL importer automatically adds code to register it with the invocation registry. For more information about interfaces and the invocation registry, see Understanding invokable interfaces (see page 2292).
- The *TSOAPConnection* component must have its UseSOAPAdapter property set to *True*. This means that the server must support the *IAppServerSOAP* interface. If the application server is built using Delphi 6 or Kylix 1, it does not support *IAppServerSOAP* and you must use a separate *THTTPRio* component instead. For details on how to call an interface using a *THTTPRio* component, see Calling invokable interfaces (see page 2307).
- You must set the SOAPServerIID property of the SOAP connection component to the GUID of the server interface. You must set this property before your application connects to the server, because it tells the *TSOAPConnection* component what interface to fetch from the server.

Assuming the previous three conditions are met, you can fetch the server interface as follows:

```
with MyConnection.GetSOAPServer as IMyAppServer do
SpecialMethod(x,y);
IDispatch* disp = (IDispatch*)(MyConnection->AppServer)
IMyAppServerDisp TempInterface( (IMyAppServer*)disp);
TempInterface.SpecialMethod(x,y);
```

See Also

Connecting to the Server (see page 1519)

Dropping or Changing a Server Connection (see page 1531)

Connecting to an Application Server That Uses Multiple Data Modules (see page 1526)

3.2.3.3.10 Choosing a Connection Protocol

Each communications protocol you can use to connect your client applications to the application server provides its own unique benefits. Before choosing a protocol, consider how many clients you expect, how you are deploying your application, and future

development plans.

The following topics describe the unique features for each connection protocol:

- Using DCOM Connections (see page 1546)
- Using Socket Connections (see page 1548)
- Using Web Connections (see page 1550)
- Using SOAP Connections (see page 1548)

See Also

Overview of AMIDAS Based MultiTiered Applications (see page 1542)

The Structure of the Client Application (see page 1543)

The Structure of the Application Server (see page 1542)

Pooling Application Servers (see page 1535)

3.2.3.3.11 Configuring TMTSDataModule

To add a *TMTSDataModule* component to your application, choose **File ▸ New ▸ Other** and select Transactional Data Module from the Multitier page of the new items dialog. You will see the Transactional Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TMTSDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TMTSDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note: You can add your own properties and methods to your new interface. For more information, see Extending the application server's interface (see page 1531).

You must specify the threading model in the Transactional Data Module wizard. Choose Single, Apartment, or Both.

- If you choose Single, client requests are serialized so that your application services only one at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment, the system ensures that any instance of your remote data module services one request at a time, and calls always use the same thread. You must guard against thread conflicts if you use global variables or objects not contained in the remote data module. Instead of using global variables, you can use the shared property manager.
- If you choose Both, MTS calls into the remote data module's interface in the same way as when you choose Apartment. However, any callbacks you make to client applications are serialized, so that you don't need to worry about them interfering with each other.

  Note: The Apartment model under MTS or COM+ is different from the corresponding model under DCOM.

  You must also specify the transaction attributes of your remote data module. You can choose from the following options:

- Requires a transaction. When you select this option, every time a client uses your remote data module's interface, that call is executed in the context of a transaction. If the caller supplies a transaction, a new transaction need not be created.
- Requires a new transaction. When you select this option, every time a client uses your remote data module's interface, a new transaction is automatically created for that call.
- Supports transactions. When you select this option, your remote data module can be used in the context of a transaction, but the caller must supply the transaction when it invokes the interface.
- Does not support transactions. When you select this option, your remote data module can't be used in the context of transactions.

See Also

Configuring TSOAPDataModule (see page 1525)

3.2.3.3.12 Configuring TRemoteDataModule

To add a *TRemoteDataModule* component to your application, choose **File** ▶ **New** ▶ **Other** and select Remote Data Module from the ActiveX page of the new items dialog. You will see the Remote Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TRemoteDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TRemoteDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note: You can add your own properties and methods to the new interface. For more information, see Extending the application server's interface (see page 1531).

You must specify the threading model in the Remote Data Module wizard. You can choose Single-threaded, Apartment-threaded, Free-threaded, or Both.

- If you choose Single-threaded, COM ensures that only one client request is serviced at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment-threaded, COM ensures that any instance of your remote data module services one request at a time. When writing code in an Apartment-threaded library, you must guard against thread conflicts if you use global variables or objects not contained in the remote data module. This is the recommended model if you are using BDE-enabled datasets. (Note that you will need a session component with its *AutoSessionName* property set to *True* to handle threading issues on BDE-enabled datasets).
- If you choose Free-threaded, your application can receive simultaneous client requests on several threads. You are responsible for ensuring your application is thread-safe. Because multiple clients can access your remote data module simultaneously, you must guard your instance data (properties, contained objects, and so on) as well as global variables. This is the recommended model if you are using ADO datasets.
- If you choose Both, your library works the same as when you choose Free-threaded, with one exception: all callbacks (calls to client interfaces) are serialized for you.
- If you choose Neutral, the remote data module can receive simultaneous calls on separate threads, as in the Free-threaded model, but COM guarantees that no two threads access the same method at the same time.

If you are creating an EXE, you must also specify what type of instancing to use. You can choose Single instance or Multiple instance (Internal instancing applies only if the client code is part of the same process space.)

- If you choose Single instance, each client connection launches its own instance of the executable. That process instantiates a single instance of the remote data module, which is dedicated to the client connection.
- If you choose Multiple instance, a single instance of the application (process) instantiates all remote data modules created for clients. Each remote data module is dedicated to a single client connection, but they all share the same process space.

See Also

Configuring TSOAPDataModule (see page 1525)

3.2.3.3.13 Configuring TSOAPDataModule

To add a *TSoapDataModule* component to your application, choose **File** ▶ **New** ▶ **Other** and select SOAP Server Data Module from the WebServices page of the new items dialog. The SOAP data module wizard appears.

You must supply a class name for your SOAP data module. This is the base name of a *TSoapDataModule* descendant*that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name MyDataServer, the wizard creates a new unit declaring TMyDataServer, a descendant of TSoapDataModule, which implements IMyDataServer, a descendant of IAppServerSOAP.*

Note: *To use TSoapDataModule, the new data module should be added to a Web Service application. The IAppServerSOAP interface is an invokable interface, which is registered in the initialization section of the new unit. This allows the invoker*

component in the main Web module to forward all incoming calls to your data module.

You may want to edit the definitions of the generated interface and TSoapDataModule descendant, adding your own properties and methods. These properties and methods are not called automatically, but client applications that request your new interface by name or GUID can use any of the properties and methods that you add.

See Also

Configuring TRemoteDataModule (see page 1525)

3.2.3.3.14 Connecting to an Application Server That Uses Multiple Data Modules

If a COM-based application server uses a main "parent" remote data module and several child remote data modules, as described in Using multiple remote data modules (see page 1547), then you need a separate connection component for every remote data module on the application server. Each connection component represents the connection to a single remote data module.

While it is possible to have your client application form independent connections to each remote data module on the application server, it is more efficient to use a single connection to the application server that is shared by all the connection components. That is, you add a single connection component that connects to the "main" remote data module on the application server, and then, for each "child" remote data module, add an additional component that shares the connection to the main remote data module.

To use a single shared connection

1. For the connection to the main remote data module, add and set up a connection component as described in Connecting to the Application Server (see page 1517). The only limitation is that you can't use a SOAP connection.
2. For each child remote data module, use a TSharedConnection component.

- Set its *ParentConnection* property to the connection component you added in step 1. The *TSharedConnection* component shares the connection that this main connection establishes.
- Set its *ChildName* property to the name of the property on the main remote data module's interface that exposes the interface of the desired child remote data module.

When you assign the *TSharedConnection* component placed in step 2 as the value of a client dataset's *RemoteServer* property, it works as if you were using an entirely independent connection to the child remote data module. However, the *TSharedConnection* component uses the connection established by the component you placed in step 1.

See Also

Connecting to the Server (see page 1519)

Dropping or Changing a Server Connection (see page 1531)

Calling Server Interfaces (see page 1522)

Using Multiple Remote Data Modules (see page 1547)

3.2.3.3.15 Creating an Active Form for the Client Application

To create an Active Form for the Client Application

1. Because the client application will be deployed as an ActiveX control, you must have a Web server that runs on the same system as the client application. You can use a ready-made server such as Microsoft's Personal Web server or you can write your own using the socket components described in "Working with Sockets." (see page 2337)
2. Create the client application following the steps described in Creating the client application (see page 1528) except start by choosing **File** ▸ **New** ▸ **ActiveX** ▸ **Active Form**, rather than beginning an ordinary client project.

3. If your client application uses a data module, add a call to explicitly create the data module in the active form initialization.
4. When your client application is finished, compile the project, and select **Project ▸ Web Deployment Options**. In the Web Deployment Options dialog, you must do the following: On the Project page, specify the Target directory, the URL for the target directory, and the HTML directory. Typically, the Target directory and the HTML directory will be the same as the projects directory for your Web Server. The target URL is typically the name of the server machine. On the Additional Files page, include midas.dll with your client application.
5. Finally, select **Project ▸ WebDeploy** to deploy the client application as an active form.

Any Web browser that can run Active forms can run your client application by specifying the .HTM file that was created when you deployed the client application. This .HTM file has the same name as your client application project, and appears in the directory specified as the Target directory.

See Also

Generating an ActiveX Control Based On a VCL Form (see page 1410)

3.2.3.3.16 Creating the Application Server

You create an application server very much as you create most database applications. The major difference is that the application server uses a remote data module.

To create an application server

1. Start a new project:

- If you are using SOAP as a transport protocol, this should be a new Web Service application. Choose **File ▸ New ▸ Other**, and on the WebServices page of the new items dialog, choose SOAP Server application. Select the type of Web Server you want to use, and when prompted whether you want to define a new interface for the SOAP module, say no.
- For any other transport protocol, you need only choose **File ▸ New ▸ Application**. Save the new project.

2. Add a new remote data module to the project. From the main menu, choose **File ▸ New ▸ Other**, and on the ActiveX, Delphi Files, or WebServices page of the new items dialog, select

- **Remote Data Module** if you are creating a COM Automation server that clients access using DCOM, HTTP, or sockets.
- **Transactional Data Module** if you are creating a remote data module that runs under COM+ (or MTS). Connections can be formed using DCOM, HTTP, or sockets. However, only DCOM supports the security services.
- **SOAP Server Data Module** if you are creating a SOAP server in a Web Service application. For more detailed information about setting up a remote data module, see Setting up the remote data module (see page 1536).

  Note: Remote data modules are more than simple data modules. The SOAP data module implements an invokable interface in a Web Service application. Other data modules are COM Automation objects.

3. Place the appropriate dataset components on the data module and set them up to access the database server.
4. Place a *TDataSetProvider* component on the data module for each dataset you want to expose to clients. This provider is required for brokering client requests and packaging data. Set the *DataSet* property for each provider to the name of the dataset to access. You can set additional properties for the provider. See Using provider components (see page 1819) for more detailed information about setting up a provider. If you are working with data from XML documents, you can use a *TXMLTransformProvider* component instead of a dataset and *TDataSetProvider* component. When using *TXMLTransformProvider*, set the *XMLDataFile* property to specify the XML document from which data is provided and to which updates are applied.
5. Write application server code to implement events, shared business rules, shared data validation, and shared security. When writing this code, you may want to

- Extend the application server's interface (see page 1531) to provide additional ways for the client application to call the server.
- Provide transaction support (see page 1533) beyond the transactions automatically created when applying updates.

- Create master/detail relationships (see page 1539) between the datasets in your application server.
- Ensure your application server is stateless (see page 1540).
- Divide your application server into multiple remote data modules (see page 1547).

6. Save, compile, and register or install the application server (see page 1535).
7. If your server application does not use DCOM or SOAP, you must install the runtime software that receives client messages, instantiates the remote data module, and marshals interface calls.

- For TCP/IP sockets this is a socket dispatcher application, Scktsrvr.exe.
- For HTTP connections this is httpsrvr.dll, an ISAPI/NSAPI DLL that must be installed with your Web server.

See Also

Creating the Client Application (see page 1528)

Registering the Application Server (see page 1535)

Writing Web-based Client Applications (see page 1551)

The Structure of the Application Server (see page 1542)

3.2.3.3.17 Creating the Client Application

In most regards, creating a multi-tiered client application is similar to creating a two-tiered client that uses a client dataset to cache updates. The major difference is that a multi-tiered client uses a connection component to establish a conduit to the application server.

To create a multi-tiered client application

1. Add a new data module to the project.
2. Place a connection component on the data module. The type of connection component you add depends on the communication protocol you want to use. See The Structure of the Client Application (see page 1543) for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see Connecting to the Application Server (see page 1517).
4. Set the other connection component properties as needed for your application. For example, you might set the *ObjectBroker* property to allow the connection component to choose dynamically from several servers. For more information about using the connection components, see Managing Server Connections (see page 1533).
5. Place as many *TClientDataSet* components as needed on the data module, and set the *RemoteServer* property for each component to the name of the connection component you placed in Step 2. For a full introduction to client datasets, see Using Client Datasets (see page 1740).
6. Set the *ProviderName* property for each *TClientDataSet* component. If your connection component is connected to the application server at design time, you can choose available application server providers from the *ProviderName* property's drop-down list.
7. Continue in the same way you would create any other database application. There are a few additional features available to clients of multi-tiered applications:

- Your application may want to make direct calls to the application server. Calling Server Interfaces (see page 1522) describes how to do this.
- You may want to use the special features of client datasets that support their interaction with the provider components. These are described in Using a Client Dataset with a Provider (see page 1732).

See Also

Creating the Application Server (see page 1527)

Registering the Application Server (see page 1535)

Writing Web-based client Applications (see page 1551)

3.2.3.3.18 Creating Web Pages with an InternetExpress Page Producer

Each InternetExpress page producer generates an HTML document that appears in the browsers of your application's clients. If your application includes several separate Web documents, use a separate page producer for each of them.

The InternetExpress page producer (*TInetXPageProducer*) is a special page producer component (see page 2270). As with other page producers, you can assign it as the Producer property of an action item or call it explicitly from an OnAction event handler. For more information about using content producers with action items, see Responding to request messages with action items (see page 2278).

The InternetExpress page producer has a default template as the value of its HTMLDoc property. This template contains a set of HTML-transparent tags that the InternetExpress page producer uses to assemble an HTML document (with embedded javascript and XML) including content produced by other components. Before it can translate all of the HTML-transparent tags and assemble this document, you must indicate the location of the javascript libraries (see page 1549) used for the embedded javascript on the page. This location is specified by setting the IncludePathURL property.

You can specify the components that generate parts of the Web page using the Web page editor (see page 1549). Display the Web page editor by double-clicking the Web page component or clicking the ellipsis button next to the *WebPageItems* property in the **Object Inspector**.

The components you add in the Web page editor generate the HTML that replaces one of the HTML-transparent tags in the InternetExpress page producer's default template. These components become the value of the WebPageItems property. After adding the components in the order you want them, you can customize the template (see page 1529) to add your own HTML or change the default tags.

See Also

Using an XML Broker (see page 1544)

3.2.3.3.19 Customizing the InternetExpress Page Producer Template

The template of an InternetExpress page producer is an HTML document with extra embedded tags that your application translates dynamically. Initially, the page producer generates a default template as the value of the HTMLDoc property. This default template has the form

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

The HTML-transparent tags in the default template are translated as follows:

<#INCLUDES> generates the statements that include the javascript libraries. These statements have the form

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldom.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldb.js"> </SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlbind.js"> </SCRIPT>
```

<#STYLES> generates the statements that defines a style sheet from definitions listed in the Styles or StylesFile property of the InternetExpress page producer.

<#WARNINGS> generates nothing at runtime. At design time, it adds warning messages for problems detected while generating the HTML document. You can see these messages in the Web page editor.

<#FORMS> generates the HTML produced by the components that you add in the Web page editor. The HTML from each component is generated in the order it appears in *WebPageItems*.

<#SCRIPT> generates a block of javascript declarations that are used in the HTML generated by the components added in the Web page editor.

You can replace the default template by changing the value of *HTMLDoc* or setting the HTMLFile property. The customized HTML template can include any of the HTML-transparent tags that make up the default template. The InternetExpress page producer automatically translates these tags when you call the Content method. In addition, The InternetExpress page producer automatically translates three additional tags:

<#BODYELEMENTS> is replaced by the same HTML as results from the 5 tags in the default template. It is useful when generating a template in an HTML editor when you want to use the default layout but add additional elements using the editor.

<#COMPONENT Name=WebComponentName> is replaced by the HTML that the component named *WebComponentName* generates. This component can be one of the components added in the Web page editor, or it can be any component that supports the IWebContent interface and has the same Owner as the InternetExpress page producer.

<#DATAPACKET XMLBroker=BrokerName> is replaced with the XML data packet obtained from the XML broker specified by *BrokerName*. When, in the Web page editor, you see the HTML that the InternetExpress page producer generates, you see this tag instead of the actual XML data packet.

In addition, the customized template can include any other HTML-transparent tags that you define. When the InternetExpress page producer encounters a tag that is not one of the seven types it translates automatically, it generates an OnHTMLTag event, where you can write code to perform your own translations. For more information about HTML templates in general, see HTML templates (see page 2269).

Tip: The components that appear in the Web page editor generate static code. That is, unless the application server changes the metadata that appears in data packets, the HTML is always the same, no matter when it is generated. You can avoid the overhead of generating this code dynamically at runtime in response to every request message by copying the generated HTML in the Web page editor and using it as a template. Because the Web page editor displays a <#DATAPACKET> tag instead of the actual XML, using this as a template still allows your application to fetch data packets from the application server dynamically.

See Also

Using the Web Page Editor (see page 1549)

HTML Templates (see page 2269)

Setting Web Item Properties (see page 1536)

3.2.3.3.20 Distributing a Client Application as an ActiveX Control

The multi-tiered database architecture can be combined with ActiveX features to distribute a client application as an ActiveX control.

When you distribute your client application as an ActiveX control, create the application server (see page 1527) as you would for any other multi-tiered application.

When creating the client application, you must use an Active Form as the basis instead of an ordinary form. See Creating an Active Form for the Client Application (see page 1526) for details.

Once you have built and deployed your client application, it can be accessed from any ActiveX-enabled Web browser on another machine. For a Web browser to successfully launch your client application, the Web server must be running on the machine that has the client application.

If the client application uses DCOM to communicate between the client application and the application server, the machine with the Web browser must be enabled to work with DCOM. If the machine with the Web browser is a Windows 95 machine, it must

have installed DCOM95, which is available from Microsoft.

See Also

Building a Multi-tiered Application (see page 1520)

Building Web Applications Using InternetExpress (see page 1522)

3.2.3.3.21 Dropping or Changing a Server Connection

A connection component drops a connection to the application server when you

- set the *Connected* property to *False*.
- free the connection component. A connection object is automatically freed when a user closes the client application.
- change any of the properties that identify the application server (*ServerName*, *ServerGUID*, *ComputerName*, and so on). Changing these properties allows you to switch among available application servers at runtime. The connection component drops the current connection and establishes a new one.

  Note: Instead of using a single connection component to switch among available application servers, a client application can instead have more than one connection component, each of which is connected to a different application server.

  Before a connection component drops a connection, it automatically calls its *BeforeDisconnect* event handler, if one is provided. To perform any special actions prior to disconnecting, write a *BeforeDisconnect* handler. Similarly, after dropping the connection, the *AfterDisconnect* event handler is called. If you want to perform any special actions after disconnecting, write an *AfterDisconnect* handler.

See Also

Connecting to the Server (see page 1519)

Calling Server Interfaces (see page 1522)

Connecting to an Application Server That Uses Multiple Data Modules (see page 1526)

3.2.3.3.22 Extending the Interface of the Application Server

Client applications interact with the application server by creating or connecting to an instance of the remote data module. They use its interface as the basis of all communication with the application server.

You can add to your remote data module's interface to provide additional support for your client applications. This interface is a descendant of *IAppServer* and is created for you automatically by the wizard when you create the remote data module (see page 1536).

To add to the remote data module's interface, you can

- Choose the Add to Interface command from the Edit menu in the IDE. Indicate whether you are adding a procedure, function, or property, and enter its syntax. When you click OK, you will be positioned in the code editor on the implementation of your new interface member.
- Use the type library editor. Select the interface for your application server in the type library editor, and click the tool button for the type of interface member (method or property) that you are adding. Give your interface member a name in the Attributes page, specify parameters and type in the Parameters page, and then refresh the type library. See Working with type libraries (see page 1445) for more information about using the type library editor .

  Note: Neither of these approaches works if you are implementing *TSoapDataModule*. For *TSoapDataModule* descendants, you must edit the server interface directly.

  When you add to a COM interface, your changes are added to your unit source code and the type library file (.TLB).

  Note: You must explicitly save the TLB file by choosing Refresh in the type library editor and then saving the changes from the IDE.

  Once you have added to your remote data module's interface, locate the properties and methods that were added to your

remote data module's implementation. Add code to finish this implementation by filling in the bodies of the new methods.

If you are not writing a SOAP data module, client applications call your interface extensions using the *AppServer* property of their connection component. With SOAP data modules, they call the connection component's *GetSOAPServer* method. For more information on how to call your interface extensions, see Calling server interfaces (see page 1522).

Adding callbacks to the application server's interface

You can allow the application server to call your client application by introducing a callback. To do this, the client application passes an interface to one of the application server's methods, and the application server later calls this method as needed. However, if your extensions to the remote data module's interface include callbacks, you can't use an HTTP or SOAP-based connection. TWebConnection and TSoapConnection do not support callbacks. If you are using a socket-based connection, client applications must indicate whether they are using callbacks by setting the SupportCallbacks property. All other types of connection automatically support callbacks.

Extending a transactional application server's interface

When using transactions or just-in-time activation, you must be sure all new methods call SetComplete to indicate when they are finished. This allows transactions to complete and permits the remote data module to be deactivated.

Furthermore, you can't return any values from your new methods that allow the client to communicate directly with objects or interfaces on the application server unless they provide a safe reference. If you are using a stateless MTS data module, neglecting to use a safe reference can lead to crashes because you can't guarantee that the remote data module is active.

See Also

Managing Transactions in Multi-tiered Applications (see page 1533)

Extending the Application Servers Interface

Supporting Master/detail Relationships (see page 1539)

Supporting State Information in Remote Data Modules (see page 1540)

Setting Up the Remote Data Module (see page 1536)

Using Multiple Remote Data Modules (see page 1547)

3.2.3.3.23 Granting Permission to Access and Launch the Application Server

Requests from the InternetExpress application appear to the application server as originating from a guest account with the name IUSR_computername, where computername is the name of the system running the Web application. By default, this account does not have access or launch permission for the application server. If you try to use the Web application without granting these permissions, when the Web browser tries to load the requested page it times out with EOLE_ACCESS_ERROR.

Note: Because the application server runs under this guest account, it can't be shut down by other accounts.

To grant the Web application access and launch permissions, run DCOMCnfg.exe, which is located in the System32 directory of the machine that runs the application server.

To configure your application server

1. When you run DCOMCnfg, select your application server in the list of applications on the Applications page.
2. Click the Properties button. When the dialog changes, select the Security page.
3. Select Use Custom Access Permissions, and press the Edit button. Add the name IUSR_computername to the list of accounts with access permission, where computername is the name of the machine that runs the Web application.
4. Select Use Custom Launch Permissions, and press the Edit button. Add IUSR_computername to this list as well.
5. Click the Apply button.

See Also

Creating the Application Server (see page 1527)

Building an InternetExpress Application (see page 1521)

Using the Javascript Libraries (see page 1549)

3.2.3.3.24 Managing Server Connections

The main purpose of connection components is to locate and connect to the application server. Because they manage server connections, you can also use connection components to call the methods of the application server's interface.

The following topics describe how to use a connection component for

- Connecting to the Server (see page 1519).
- Dropping or Changing a Server Connection (see page 1531).
- Calling Server Interfaces (see page 1522).
- Connecting to an Application Server that Uses Multiple Data Modules (see page 1526).

See Also

Connecting to the Application Server (see page 1517)

The Structure of the Client Application (see page 1543)

Using a Client dataset with a data provider (see page 1732)

3.2.3.3.25 Managing Transactions in Multi-tiered Applications

When client applications apply updates to the application server, the provider component automatically wraps the process of applying updates and resolving errors in a transaction. This transaction is committed if the number of problem records does not exceed the *MaxErrors* value specified as an argument to the *ApplyUpdates* method. Otherwise, it is rolled back.

In addition, you can add transaction support to your server application by adding a database connection component or managing the transaction directly by sending SQL to the database server. This works the same way that you would manage transactions in a two-tiered application. For more information about this sort of transaction control, see Managing transactions (see page 1498).

If you have a transactional data module, you can broaden your transaction support by using COM+ (or MTS) transactions. These transactions can include any of the business logic on your application server, not just the database access. In addition, because they support two-phase commits, they can span multiple databases.

Only the BDE- and ADO-based data access components support two-phase commit. Do not use InterbaseExpress or dbExpress components if you want to have transactions that span multiple databases.

Warning: When using the BDE, two-phase commit is fully implemented only on Oracle7 and MS-SQL databases. If your transaction involves multiple databases, and some of them are remote servers other than Oracle7 or MS-SQL, your transaction runs a small risk of only partially succeeding. Within any one database, however, you will always have transaction support.

By default, all *IAppServer* calls on a transactional data module are transactional. You need only set the transaction attribute of your data module (see page 1524) to indicate that it must participate in transactions. In addition, you can extend the application server's interface (see page 1531) to include method calls that encapsulate transactions that you define.

If your transaction attribute indicates that the remote data module requires a transaction, then every time a client calls a method on its interface, it is automatically wrapped in a transaction. All client calls to your application server are then enlisted in that transaction until you indicate that the transaction is complete. These calls either succeed as a whole or are rolled back.

Note: Do not combine COM+ or MTS transactions with explicit transactions created by a database connection component or using explicit SQL commands. When your transactional data module is enlisted in a transaction, it automatically enlists all of your database calls in the transaction as well.

See Also

Managing Transactions in Multi-tiered Applications

Extending the Application Servers Interface (see page 1531)

Supporting Master/detail Relationships (see page 1539)

Supporting State Information in Remote Data Modules (see page 1540)

Setting Up the Remote Data Module (see page 1536)

Using Multiple Remote Data Modules (see page 1547)

3.2.3.3.26 Overview of a Three-tiered Application

The following numbered steps illustrate a normal sequence of events for a provider-based three-tiered application:

1. A user starts the client application. The client connects to the application server (which can be specified at design time or runtime). If the application server is not already running, it starts. The client receives an *IAppServer* interface for communicating with the application server.
2. The client requests data from the application server. A client may request all data at once, or may request chunks of data throughout the session (fetch on demand).
3. The application server retrieves the data (first establishing a database connection, if necessary), packages it for the client, and returns a data packet to the client. Additional information, (for example, field display characteristics) can be included in the metadata of the data packet. This process of packaging data into data packets is called "providing."
4. The client decodes the data packet and displays the data to the user.
5. As the user interacts with the client application, the data is updated (records are added, deleted, or modified). These modifications are stored in a change log by the client.
6. Eventually the client applies its updates to the application server, usually in response to a user action. To apply updates, the client packages its change log and sends it as a data packet to the server.
7. The application server decodes the package and posts updates (in the context of a transaction if appropriate). If a record can't be posted (for example, because another application changed the record after the client requested it and before the client applied its updates), the application server either attempts to reconcile the client's changes with the current data, or saves the records that could not be posted. This process of posting records and caching problem records is called "resolving."
8. When the application server finishes the resolving process, it returns any unposted records to the client for further resolution.
9. The client reconciles unresolved records. There are many ways a client can reconcile unresolved records. Typically the client attempts to correct the situation that prevented records from being posted or discards the changes. If the error situation can be rectified, the client applies updates again.
10. The client refreshes its data from the server.

See Also

The Structure of the Client Application (see page 1543)

The Structure of the Application Server (see page 1542)

Choosing a Connection Protocol (see page 1523)

Using a Multi-tiered Architecture (see page 1564)

3.2.3.3.27 Pooling Remote Data Modules

Object pooling allows you to create a cache of remote data modules that are shared by their clients, thereby conserving resources. How this works depends on the type of remote data module and on the connection protocol (see page 1523).

If you are creating a transactional data module (see page 1546) that will be installed to COM+, you can use the COM+ Component Manager to install the application server as a pooled object.

Even if you are not using a transactional data module, you can take advantage of object pooling if the connection is formed using *TWebConnection*. Under this second type of object pooling, you limit the number of instances of your remote data module that are created. This limits the number of database connections that you must hold, as well as any other resources used by the remote data module.

When the Web Server application (which passes calls to your remote data module) receives client requests, it passes them on to the first available remote data module in the pool. If there is no available remote data module, it creates a new one (up to a maximum number that you specify). This provides a middle ground between routing all clients through a single remote data module instance (which can act as a bottleneck), and creating a separate instance for every client connection (which can consume many resources).

If a remote data module instance in the pool does not receive any client requests for a while, it is automatically freed. This prevents the pool from monopolizing resources unless they are used.

To set up object pooling when using a Web connection (HTTP), your remote data module must override the UpdateRegistry method. In the overridden method, call RegisterPooled when the remote data module registers and UnregisterPooled when the remote data module unregisters.

When using either method of object pooling, your remote data module must be stateless. This is because a single instance potentially handles requests from several clients. If it relied on persistent state information, clients could interfere with each other. See Supporting State Information in Remote Data Modules (see page 1540) for more information on how to ensure that your remote data module is stateless.

See Also

Choosing a Connection Protocol (see page 1523)

3.2.3.3.28 Registering the Application Server

Before client applications can locate and use an application server, it must be registered or installed.

- If the application server uses DCOM, HTTP, or sockets as a communication protocol, it acts as an Automation server and must be registered like any other COM server. For information about registering a COM server, see Registering a COM Object (see page 1440).
- If you are using a transactional data module, you do not register the application server. Instead, you install it with COM+ or MTS. .
- When the application server uses SOAP, the application must be a Web Service application. As such, it must be registered with your Web Server, so that it receives incoming HTTP messages. In addition, you need to publish a WSDL document that describes the invokable interfaces in your application. For information about exporting a WSDL document for a Web Service application, see Generating WSDL Documents for a Web Service Application (see page 2306).

See Also

Creating the Client Application (see page 1528)

Creating the Application Server (see page 1527)

Writing Web-based client applications (see page 1551)

3.2.3.3.29 Setting Up the Remote Data Module

When you create the remote data module, you must provide certain information that indicates how it responds to client requests. This information varies, depending on the type of remote data module. See The Structure of the Application Server (see page 1542) for information on what type of remote data module you need.

The following topics describe how to configure each type of remote data module:

- Configuring TRemoteDataModule (see page 1525)
- Configuring TMTSDataModule (see page 1524)
- Configuring TSoapDataModule (see page 1525)

See Also

Managing Transactions in Multi-tiered Applications (see page 1533)

Extending the Application Servers Interface (see page 1531)

Supporting Master/detail Relationships (see page 1539)

Supporting State Information in Remote Data Modules (see page 1540)

Using Multiple Remote Data Modules (see page 1547)

3.2.3.3.30 Setting Web Item Properties

The Web items that you add using the Web page editor are specialized components that generate HTML. Each Web item class is designed to produce a specific control or section of the final HTML document, but a common set of properties influences the appearance of the final HTML.

When a Web item represents information from the XML data packet (for example, when it generates a set of field or parameter display controls or a button that manipulates the data), the XMLBroker property associates the Web item with the XML broker that manages the data packet. You can further specify a dataset that is contained in a dataset field of that data packet using the XMLDataSetField property. If the Web item represents a specific field or parameter value, the Web item has a FieldName or ParamName property.

You can apply a style attribute to any Web item, thereby influencing the overall appearance of all the HTML it generates. Styles and style sheets are part of the HTML 4 standard. They allow an HTML document to define a set of display attributes that apply to a tag and everything in its scope. Web items offer a flexible selection of ways to use them:

The simplest way to use styles is to define a style attribute directly on the Web item. To do this, use the Style property. The value of *Style* is simply the attribute definition portion of a standard HTML style definition, such as,

```
color: red.
```

You can define a style sheet that defines a set of style definitions. Each definition includes a style selector (the name of a tag to which the style always applies or a user-defined style name) and the attribute definition in curly braces,

```
H2 B  {color: red}
.MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
```

The entire set of definitions is maintained by the InternetExpress page producer as its Styles property. Each Web item can then reference the styles with user-defined names by setting its StyleRule property.

If you are sharing a style sheet with other applications, you can also supply the style definitions as the value of the InternetExpress page producer's StylesFile property instead of the *Styles* property. Individual Web items still reference styles using the *StyleRule* property.

Another common property of Web items is the *Custom* property. *Custom* includes a set of options that you add to the generated

HTML tag. HTML defines a different set of options for each type of tag. The VCL reference for the *Custom* property of most Web items gives an example of possible options. For more information on possible options, use an HTML reference.

See Also

Using the Web Page Editor (see page 1549)

Customizing the InternetExpress Page Producer Template (see page 1529)

3.2.3.3.31 Specifying a Connection Using HTTP

You can establish a connection to the application server using HTTP from any machine that has a TCP/IP address. Unlike sockets, however, HTTP allows you to take advantage of SSL security and to communicate with a server that is protected behind a firewall. When using HTTP, include a TWebConnection component for connecting to the application server.

The Web connection component establishes a connection to the Web server application (httpsrvr.dll), which in turn communicates with the application server. *TWebConnection* locates httpsrvr.dll using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server and httpsrvr.dll, and the path to the Web server application (httpsrvr.dll). Specify this value using the URL property.

Note: When using *TWebConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

If the Web server requires authentication, or if you are using a proxy server that requires authentication, you must set the values of the UserName and Password properties so that the connection component can log on.

If you have multiple servers that your client application can choose from, you can use the ObjectBroker property instead of specifying a value for URL. For more information, see Brokering connections (see page 1520).

See Also

Specifying a Connection Using DCOM (see page 1537)

Specifying a Connection Using Sockets (see page 1538)

Using Web Connections (see page 1550)

Parts of a Uniform Resource Locator (see page 2253)

3.2.3.3.32 Specifying a Connection Using DCOM

When using DCOM to communicate with the application server, client applications include a TDCOMConnection component for connecting to the application server. *TDCOMConnection* uses the ComputerName property to identify the machine on which the server resides.

When *ComputerName* is blank, the DCOM connection component assumes that the application server resides on the client machine or that the application server has a system registry entry. If you do not provide a system registry entry for the application server on the client when using DCOM, and the server resides on a different machine from the client, you must supply *ComputerName*.

Note: Even when there is a system registry entry for the application server, you can specify *ComputerName* to override this entry. This can be especially useful during development, testing, and debugging.

If you have multiple servers that your client application can choose from, you can use the ObjectBroker property instead of specifying a value for *ComputerName*. For more information, see Brokering connections (see page 1520).

If you supply the name of a host computer or server that cannot be found, the DCOM connection component raises an exception when you try to open the connection.

See Also

Specifying a Connection Using Sockets (see page 1538)

Using DCOM Connections (see page 1546)

Specifying a Connection Using HTTP (see page 1537)

3.2.3.3.33 Specifying a Connection Using SOAP

You can establish a connection to a SOAP application server using the TSoapConnection component. *TSoapConnection* is very similar to *TWebConnection*, because it also uses HTTP as a transport protocol. Thus, you can use *TSoapConnection* from any machine that has a TCP/IP address, and it can take advantage of SSL security and to communicate with a server that is protected by a firewall.

The SOAP connection component establishes a connection to a Web Service provider that implements the *IAppServerSOAP* or *IAppServer* interface. (The UseSOAPAdapter property specifies which interface it expects the server to support.) If the server implements the *IAppServerSOAP* interface, *TSoapConnection* converts that interface to an *IAppServer* interface for client datasets. *TSoapConnection* locates the Web Server application using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server, the name of the Web Service application, and a path that matches the path name of the *THTTPSoapDispatcher* on the application server. Specify this value using the URL property.

By default, *TSOAPConnection* automatically looks for an *IAppServerSOAP* (or *IAppServer*) interface. If the server includes more than one remote data module, you must indicate the target data module's interface (an *IAppServerSOAP* descendant) so that *TSOAPConnection* can identify the remote data module you want to use. There are two ways to do this:

- Set the SOAPServerIID property to indicate the interface of the target remote data module. This method works for any server that implements an *IAppServerSOAP* descendant. *SOAPServerIID* identifies the target interface by its GUID. At runtime, you can use the interface name, and the compiler automatically extracts the GUID. However, at design time, in the **Object Inspector**, you must specify the GUID string.
- If the server is written using the Delphi language, you can simply include the name of the SOAP data module's interface following a slash at the end of the path portion of the URL. This lets you specify the interface by name rather than GUID, but is only available when both client and server are written in Delphi.

  Tip: The first approach, using the *SOAPServerIID* method, has the added advantage that it lets you call extensions to the remote data module's interface (see page 1522).

  If you are using a proxy server, you must indicate the name of the proxy server using the Proxy property. If that proxy requires authentication, you must also set the values of the UserName and Password properties so that the connection component can log on.

  Note: When using *TSoapConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

See Also

Specifying a Connection Using DCOM (see page 1537)

Specifying a Connection Using Sockets (see page 1538)

3.2.3.3.34 Specifying a Connection Using Sockets

You can establish a connection to the application server using sockets from any machine that has a TCP/IP address. This method has the advantage of being applicable to more machines, but does not provide for using any security protocols. When using sockets, include a TSocketConnection component for connecting to the application server.

TSocketConnection identifies the server machine using the IP Address or host name of the server system, and the port number

of the socket dispatcher program (Scktsrvr.exe) that is running on the server machine. For more information about IP addresses and port values, see Describing sockets (see page 2336).

Three properties of *TSocketConnection* specify this information:

- Address specifies the IP Address of the server.
- Host specifies the host name of the server.
- Port specifies the port number of the socket dispatcher program on the application server.

Address and *Host* are mutually exclusive. Setting one unsets the value of the other. For information on which one to use, see Describing the host (see page 2336).

If you have multiple servers that your client application can choose from, you can use the ObjectBroker property instead of specifying a value for *Address* or *Host*. For more information, see Brokering connections (see page 1520).

By default, the value of *Port* is 211, which is the default port number of the socket dispatcher program that forwards incoming messages to your application server. If the socket dispatcher has been configured to use a different port, set the *Port* property to match that value.

Note: You can configure the port of the socket dispatcher while it is running by right-clicking the CodeGear Socket Server tray icon and choosing Properties.

Although socket connections do not provide for using security protocols, you can customize the socket connection to add your own encryption.

To add your own encryption

1. Create a COM object that supports the IDataIntercept interface. This is an interface for encrypting and decrypting data.
2. Use TPacketInterceptFactory as the class factory for this object. If you are using a wizard to create the COM object in step 1, replace the line in the initialization section that says `TComponentFactory.Create(...)` with `TPacketInterceptFactory.Create(...)`.
3. Register your new COM server on the client machine.
4. Set the InterceptName or InterceptGUID property of the socket connection component to specify this COM object. If you used *TPacketInterceptFactory* in step 2, your COM server appears in the drop-down list of the **Object Inspector** for the *InterceptName* property.
5. Finally, right click the CodeGear Socket Server tray icon, choose Properties, and on the properties tab set the Intercept Name or Intercept GUID to the ProgId or GUID for the interceptor.

This mechanism can also be used for data compression and decompression.

See Also

Specifying a Connection Using DCOM (see page 1537)

Using Socket Connections (see page 1548)

Working with Sockets (see page 2337)

Specifying a Connection Using HTTP (see page 1537)

3.2.3.3.35 Supporting Master/detail Relationships

You can create master/detail relationships between client datasets in your client application in the same way you set them up using any table-type dataset. For more information about setting up master/detail relationships in this way, see Creating Master/detail Relationships (see page 1590).

However, this approach has two major drawbacks:

- The detail table must fetch and store all of its records from the application server even though it only uses one detail set at a time. (This problem can be mitigated by using parameters. For more information, see Limiting records with parameters (see page 1720).)

- It is very difficult to apply updates, because client datasets apply updates at the dataset level and master/detail updates span multiple datasets. Even in a two-tiered environment, where you can use the database connection component to apply updates for multiple tables in a single transaction, applying updates in master/detail forms is tricky.

In multi-tiered applications, you can avoid these problems by using nested tables to represent the master/detail relationship. To do this when providing from datasets, set up a master/detail relationship between the datasets on the application server. Then set the *DataSet* property of your provider component to the master table. To use nested tables to represent master/detail relationships when providing from XML documents, use a transformation file that defines the nested detail sets.

When clients call the *GetRecords* method of the provider, it automatically includes the detail dataset as a DataSet field in the records of the data packet. When clients call the *ApplyUpdates* method of the provider, it automatically handles applying updates in the proper order.

See Also

Managing Transactions in Multi-tiered Applications (see page 1533)

Extending the Application Servers Interface (see page 1531)

Supporting Master/detail Relationships

Supporting State Information in Remote Data Modules (see page 1540)

Setting Up the Remote Data Module (see page 1536)

Using Multiple Remote Data Modules (see page 1547)

3.2.3.3.36 Supporting State Information in Remote Data Modules

The *IAppServer* interface, which client datasets use to communicate with providers on the application server, is mostly stateless. When an application is stateless, it does not "remember" anything that happened in previous calls by the client. This stateless quality is useful if you are pooling database connections in a transactional data module, because your application server does not need to distinguish between database connections for persistent information such as record currency. Similarly, this stateless quality is important when you are sharing remote data module instances between many clients, as occurs with just-in-time activation or object pooling. SOAP data modules must be stateless.

However, there are times when you want to maintain state information between calls to the application server. For example, when requesting data using incremental fetching (see page 1725), the provider on the application server must "remember" information from previous calls (the current record).

Before and after any calls to the *IAppServer* interface that the client dataset makes (*AS_ApplyUpdates*, *AS_Execute*, *AS_GetParams*, *AS_GetRecords*, or *AS_RowRequest*), it receives an event where it can send or retrieve custom state information. Similarly, before and after providers respond to these client-generated calls, they receive events where they can retrieve or send custom state information. Using this mechanism, you can communicate persistent state information between client applications and the application server, even if the application server is stateless.

For example, consider a dataset that represents the following parameterized query:

```
SELECT * from CUSTOMER WHERE CUST_NO > :MinVal ORDER BY CUST_NO
```

To enable incremental fetching in a stateless application server, you can do the following:

When the provider packages a set of records in a data packet, it notes the value of CUST_NO on the last record in the packet:

```
TRemoteDataModule1.DataSetProvider1GetData(Sender: TObject; DataSet: TCustomClientDataSet);
begin
  DataSet.Last; { move to the last record }
  with Sender as TDataSetProvider do
    Tag := DataSet.FieldValues['CUST_NO']; {save the value of CUST_NO }
end;
TRemoteDataModule1::DataSetProvider1GetData(TObject *Sender, TCustomClientDataSet *DataSet)
{
  DataSet->Last(); // move to the last record
```

```
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  pProvider->Tag = DataSet->FieldValues["CUST_NO"];
}
```

The provider sends this last CUST_NO value to the client after sending the data packet:

```
TRemoteDataModule1.DataSetProvider1AfterGetRecords(Sender: TObject;
                  var OwnerData: OleVariant);
begin
  with Sender as TDataSetProvider do
    OwnerData := Tag; {send the last value of CUST_NO }
end;
TRemoteDataModule1::DataSetProvider1AfterGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TComponent *pProvider = dynamic_cast<TComponent *>(Sender);
  OwnerData = pProvider->Tag;
}
```

On the client, the client dataset saves this last value of CUST_NO:

```
TDataModule1.ClientDataSet1AfterGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
    Tag := OwnerData; {save the last value of CUST_NO }
end;
TDataModule1::ClientDataSet1AfterGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TComponent *pDS = dynamic_cast<TComponent *>(Sender);
  pDS->Tag = OwnerData;
}
```

Before fetching a data packet, the client sends the last value of CUST_NO it received:

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData: OleVariant);
begin
  with Sender as TClientDataSet do
  begin
    if not Active then Exit;
    OwnerData := Tag; { Send last value of CUST_NO to application server }
  end;
end;
TDataModule1::ClientDataSet1BeforeGetRecords(TObject *Sender, OleVariant &OwnerData)
{
  TClientDataSet *pDS = dynamic_cast<TClientDataSet *>(Sender);
  if (!pDS->Active)
    return;
  OwnerData = pDS->Tag;
}
```

Finally, on the server, the provider uses the last CUST_NO sent as a minimum value in the query:

```
TRemoteDataModule1.DataSetProvider1BeforeGetRecords(Sender: TObject;
                   var OwnerData: OleVariant);
begin
  if not VarIsEmpty(OwnerData) then
    with Sender as TDataSetProvider do
      with DataSet as TSQLDataSet do
      begin
        Params.ParamValues['MinVal'] := OwnerData;
        Refresh; { force the query to reexecute }
    end;
end;
TRemoteDataModule1::DataSetProvider1BeforeGetRecords(TObject *Sender, OleVariant &OwnerData)
{
if (!VarIsEmpty(OwnerData))
{
  TDataSetProvider *pProv = dynamic_cast<TDataSetProvider *>(Sender);
  TSQLDataSet *pDS = (dynamic_cast<TSQLDataSet *>(pProv->DataSet);
```

```
  pDS->Params->ParamValues["MinVal"] = OwnerData;
  pDS->Refresh(); // force the query to reexecute
}
}
```

See Also

Managing Transactions in Multi-tiered Applications (see page 1533)

Extending the Application Servers Interface (see page 1531)

Supporting Master/detail Relationships (see page 1539)

Supporting State Information in Remote Data Modules

Setting Up the Remote Data Module (see page 1536)

Writing Web-based Client Applications (see page 1551)

Communicating with The Client Dataset (see page 1811)

Using Multiple Remote Data Modules (see page 1547)

3.2.3.3.37 The Structure of the Application Server

When you set up and run an application server, it does not establish any connection with client applications. Rather, client applications initiate and maintain the connection. The client application uses a connection component to connect to the application server, and uses the interface of the application server to communicate with a selected provider. All of this happens automatically, without your having to write code to manage incoming requests or supply interfaces.

The basis of an application server is a remote data module, which is a specialized data module that supports the *IAppServer* interface (for application servers that also function as a Web Service, the remote data module supports the *IAppServerSOAP* interface as well, and uses it in preference to *IAppServer*.) Client applications use the remote data module's interface to communicate with providers (see page 1811) on the application server. When the remote data module uses *IAppServerSOAP*, the connection component adapts this to an *IAppServer* interface that client datasets can use.

There are three types of remote data modules:

- **TRemoteDataModule**: This is a dual-interface Automation server. Use this type of remote data module if clients use DCOM, HTTP, sockets, or OLE to connect to the application server, unless you want to install the application server with COM+.
- **TMTSDataModule**: This is a dual-interface Automation server. Use this type of remote data module if you are creating the application server as an Active Library (.DLL) that is installed with COM+ (or MTS). You can use MTS remote data modules with DCOM, HTTP, sockets, or OLE. See Using transactional data modules (see page 1546) for information about the benefits and limitations of using MTS or COM+ with the application server.
- **TSoapDataModule**: This is a data module that implements an *IAppServerSOAP* interface in a Web Service application. Use this type of remote data module to provide data to clients that access data as a Web Service.

  Note: If the application server is to be deployed under COM+ (or MTS), the remote data module includes events for when the application server is activated or deactivated. This allows it to acquire database connections when activated and release them when deactivated.

The contents of the remote data module

As with any data module, you can include any nonvisual component in the remote data module. There are certain components, however, that you must include:

It must include a dataset component to represent the records from that database server if the remote data module is exposing information from a database server. Other components, such as a database connection component (see page 1506) of some type, may be required to allow the dataset to interact with a database server.

For every dataset that the remote data module exposes to clients, it must include a dataset provider (see page 1819). A

dataset provider packages data into data packets that are sent to client datasets and applies updates received from client datasets back to a source dataset or a database server.

It must include an XML provider (see page 1846) for every XML document that the remote data module exposes to clients. An XML provider acts like a dataset provider, except that it fetches data from and applies updates to an XML document rather than a database server.

Note: Do not confuse database connection components, which connect datasets to a database server, with the connection components used by client applications in a multi-tiered application. The connection components in multi-tiered applications can be found on the DataSnap category or WebServices category of the Tool palette

.

See Also

Overview of AMIDAS Based MultiTiered Applications

The Structure of the Client Application (see page 1543)

Creating the Application Server (see page 1527)

3.2.3.3.38 The Structure of the Client Application

To the end user, the client application of a multi-tiered application looks and behaves no differently than a two-tiered application that uses cached updates. User interaction takes place through standard data-aware controls that display data from a TClientDataSet component. For detailed information about using the properties, events, and methods of client datasets, see Using Client Datasets (see page 1740).

TClientDataSet fetches data from and applies updates to a provider component, just as in two-tiered applications that use a client dataset with an external provider. For details about providers, see Using Provider Components (see page 1819). For details about client dataset features that facilitate its communication with a provider, see Using a Client Dataset with a Provider (see page 1732).

The client dataset communicates with the provider through the IAppServer interface. It gets this interface from a connection component. The connection component establishes the connection to the application server. Different connection components are available for using different communications protocols (see page 1523).

These connection components are summarized in the following table:

Connection components

Component	Protocol
TDCOMConnection	DCOM
TSocketConnection	Windows sockets (TCP/IP)
TWebConnection	HTTP
TSOAPConnection	SOAP (HTTP and XML)

Note: The DataSnap category of the Tool palette

also includes a connection component that does not connect to an application server at all, but instead supplies an IAppServer interface for client datasets to use when communicating with providers in the same application. This component, TLocalConnection, is not required, but makes it easier to scale up to a multi-tiered application later. For more information about using connection components, see Connecting to the Application Server (see page 1517).

See Also

Overview of a Three-tiered Application (see page 1534)

The Structure of the Application Server (see page 1542)

Creating the Client Application (see page 1528)

3.2.3.3.39 Understanding Multi-tiered Database Applications

Multi-tiered applications use the components on the DataSnap page, the Data Access page, and possibly the WebServices page of the **Tool palette**, plus a remote data module that is created by a wizard on the Multitier or WebServices page of the New Items dialog. They are based on the ability of provider components to package data into transportable data packets and handle updates received as transportable delta packets.

The components needed for a multi-tiered application are described in the following table:

Components used in multi-tiered applications

Component	Description
Remote data modules	Specialized data modules that can act as a COM Automation server or implement a Web Service to give client applications access to any providers they contain. Used on the application server.
Provider component	A data broker that provides data by creating data packets and resolves client updates. Used on the application server.
Client dataset component	A specialized dataset that uses midas.dll or midaslib.dcu to manage data stored in data packets. The client dataset is used in the client application. It caches updates locally, and applies them in delta packets to the application server.
Connection components	A family of components that locate the server, form connections, and make the *IAppServer* interface available to client datasets. Each connection component is specialized to use a particular communications protocol.

The provider and client dataset components require midas.dll or midaslib.dcu, which manages datasets stored as data packets. (Note that, because the provider is used on the application server and the client dataset is used on the client application, if you are using midas.dll, you must deploy it on both application server and client application.)

An overview of the architecture into which these components fit is described in Using a multi-tiered architecture (see page 1564). For more information on how these components fit together to create a multi-tiered application, see

- Overview of a Three-tiered Application (see page 1534)
- The Structure of the Client Application (see page 1543)
- The Structure of the Application Server (see page 1542)
- Choosing a Connection Protocol (see page 1523)

See Also

Advantages of the Multi-tiered Database Model (see page 1519)

Building a Multi-tiered Application (see page 1520)

Writing Web-based Client Applications (see page 1551)

3.2.3.3.40 Using an XML Broker

The XML broker serves two major functions:

- It fetches XML data packets from the application server and makes them available to the Web Items that generate HTML for the InternetExpress application.
- It receives updates in the form of XML delta packets from browsers and applies them to the application server.

Fetching XML data packets

Before the XML broker can supply XML data packets to the components that generate HTML pages, it must fetch them from the application server. To do this, it uses the *IAppServer* interface, which it acquires from a connection component.

Note: Even when using SOAP, where the application server supports *IAppServerSOAP*, the XML broker uses *IAppServer* because the connection component acts as an adapter between the two interfaces.

You must set the following properties so that the XML producer can use the *IAppServer* interface:

- Set the RemoteServer property to the connection component that establishes the connection to the application server and gets its *IAppServer* interface. At design time, you can select this value from a drop-down list in the **Object Inspector**.
- Set the ProviderName property to the name of the provider component on the application server that represents the dataset for which you want XML data packets. This provider both supplies XML data packets and applies updates from XML delta packets. At design time, if the *RemoteServer* property is set and the connection component has an active connection, the **Object Inspector** displays a list of available providers. (If you are using a DCOM connection the application server must also be registered on the client machine).

Two properties let you indicate what you want to include in data packets:

- You can limit the number of records that are added to the data packet by setting the MaxRecords property. This is especially important for large datasets because InternetExpress applications send the entire data packet to client Web browsers. If the data packet is too large, the download time can become prohibitively long.
- If the provider on the application server represents a query or stored procedure, you may want to provide parameter values before obtaining an XML data packet. You can supply these parameter values using the Params property.

The components that generate HTML and javascript for the InternetExpress application automatically use the XML broker's XML data packet once you set their *XMLBroker* property. To obtain the XML data packet directly in code, use the RequestRecords method.

Note: When the XML broker supplies a data packet to another component (or when you call *RequestRecords*), it receives an OnRequestRecords event. You can use this event to supply your own XML string instead of the data packet from the application server. For example, you could fetch the XML data packet from the application server using GetXMLRecords and then edit it before supplying it to the emerging Web page.

Applying updates from XML delta packets

When you add the XML broker to the Web module (or data module containing a *TWebDispatcher*), it automatically registers itself with the Web dispatcher as an auto-dispatching object. This means that, unlike other components, you do not need to create an action item for the XML broker in order for it to respond to update messages from a Web browser. These messages contain XML delta packets that should be applied to the application server. Typically, they originate from a button that you create on one of the HTML pages produced by the Web client application.

So that the dispatcher can recognize messages for the XML broker, you must describe them using the WebDispatch property. Set the *PathInfo* property to the path portion of the URL to which messages for the XML broker are sent. Set *MethodType* to the value of the method header of update messages addressed to that URL (typically *mtPost*). If you want to respond to all messages with the specified path, set *MethodType* to *mtAny*. If you don't want the XML broker to respond directly to update messages (for example, if you want to handle them explicitly using an action item), set the *Enabled* property to *False*. For more information on how the Web dispatcher determines which component handles messages from the Web browser, see Dispatching request messages (see page 2281).

When the dispatcher passes an update message on to the XML broker, it passes the updates on to the application server and, if there are update errors, receives an XML delta packet describing all update errors. Finally, it sends a response message back to the browser, which either redirects the browser to the same page that generated the XML delta packet or sends it some new content.

A number of events allow you to insert custom processing at all steps of this update process:

1. When the dispatcher first passes the update message to the XML broker, it receives a BeforeDispatch event, where you can preprocess the request or even handle it entirely. This event allows the XML broker to handle messages other than update messages.

2. If the *BeforeDispatch* event handler does not handle the message, the XML broker receives an OnRequestUpdate event, where you can apply the updates yourself rather than using the default processing.
3. If the *OnRequestUpdate* event handler does not handle the request, the XML broker applies the updates and receives a delta packet containing any update errors.
4. If there are no update errors, the XML broker receives an OnGetResponse event, where you can create a response message that indicates the updates were successfully applied or sends refreshed data to the browser. If the *OnGetResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker sends a response that redirects the browser back to the document that generated the delta packet.
5. If there are update errors, the XML broker receives an OnGetErrorResponse event instead. You can use this event to try to resolve update errors or to generate a Web page that describes them to the end user. If the *OnGetErrorResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker calls on a special content producer called the ReconcileProducer to generate the content of the response message.
6. Finally, the XML broker receives an AfterDispatch event, where you can perform any final actions before sending a response back to the Web browser.

See Also

Creating Web Pages with an InternetExpress Page Producer (see page 1529)

3.2.3.3.41 Using DCOM Connections

DCOM provides the most direct approach to communication, requiring no additional runtime applications on the server.

DCOM provides the only approach that lets you use security services when writing a transactional data module. These security services are based on assigning roles to the callers of transactional objects. When using DCOM, DCOM identifies the caller to the system that calls your application server (COM+ or MTS). Therefore, it is possible to accurately determine the role of the caller. When using other protocols, however, there is a runtime executable, separate from the application server, that receives client calls. This runtime executable makes COM calls into the application server on behalf of the client. Because of this, it is impossible to assign roles to separate clients: The runtime executable is, effectively, the only client.

See Also

Using Transactional Data Modules (see page 1546)

Using Socket Connections (see page 1548)

Using SOAP Connections (see page 1548)

3.2.3.3.42 Using Transactional Data Modules

You can write an application server that takes advantage of special services for distributed applications that are supplied by COM+ (under Windows 2000 and later) or MTS (before Windows 2000). To do so, create a transactional data module instead of an ordinary remote data module.

When you use a transactional data module, your application can take advantage of the following special services:

- **Security.** COM+ (or MTS) provides role-based security for your application server. Clients are assigned roles, which determine how they can access the MTS data module's interface. The MTS data module implements the IsCallerInRole method, which you lets you check the role of the currently connected client and conditionally allow certain functions based on that role. .
- **Database handle pooling.** Transactional data modules automatically pool database connections that are made via ADO or (if you are using MTS and turn on MTS POOLING) the BDE. When one client is finished with a database connection, another client can reuse it. This cuts down on network traffic, because your middle tier does not need to log off of the remote database server and then log on again. When pooling database handles, your database connection component should set its *KeepConnection* property to *False*, so that your application maximizes the sharing of connections. .
- **Transactions**. When using a transactional data module, you can provide enhanced transaction support beyond that available

with a single database connection. Transactional data modules can participate in transactions that span multiple databases, or include functions that do not involve databases at all. For more information about the transaction support provided by transactional objects such as transactional data modules, see Managing transactions in multi-tiered applications (see page 1533).

- **Just-in-time activation and as-soon-as-possible deactivation.** You can write your server so that remote data module instances are activated and deactivated on an as-needed basis. When using just-in-time activation and as-soon-as-possible deactivation, your remote data module is instantiated only when it is needed to handle client requests. This prevents it from tying up resources such as database handles when they are not in use.

Using just-in-time activation and as-soon-as-possible deactivation provides a middle ground between routing all clients through a single remote data module instance, and creating a separate instance for every client connection. With a single remote data module instance, the application server must handle all database calls through a single database connection. This acts as a bottleneck, and can impact performance when there are many clients. With multiple instances of the remote data module, each instance can maintain a separate database connection, thereby avoiding the need to serialize database access. However, this monopolizes resources because other clients can't use the database connection while it is associated with another client's remote data module.

To take advantage of transactions, just-in-time activation, and as-soon-as-possible deactivation, remote data module instances must be stateless. This means you must provide additional support if your client relies on state information. For example, the client must pass information about the current record when performing incremental fetches. For more information about state information and remote data modules in multi-tiered applications, see Supporting state information in remote data modules (see page 1540).

By default, all automatically generated calls to a transactional data module are transactional (that is, they assume that when the call exits, the data module can be deactivated and any current transactions committed or rolled back). You can write a transactional data module that depends on persistent state information by setting the AutoComplete property to *False*, but it will not support transactions, just-in-time activation, or as-soon-as-possible deactivation unless you use a custom interface.

Warning: Application servers containing transactional data modules should not open database connections until the data module is activated. While developing your application, be sure that all datasets are not active and the database is not connected before running your application. In the application itself, add code to open database connections when the data module is activated and close them when it is deactivated.

See Also

Choosing a Connection Protocol (see page 1523)

Pooling Remote Data Modules (see page 1535)

3.2.3.3.43 Using Multiple Remote Data Modules

You may want to structure your application server so that it uses multiple remote data modules. Using multiple remote data modules lets you partition your code, organizing a large application server into multiple units, where each unit is relatively self-contained.

Although you can always create multiple remote data modules on the application server that function independently, a special connection component on the DataSnap category of the **Tool palette** provides support for a model where you have one main "parent" remote data module that dispatches connections from clients to other "child" remote data modules. This model requires that you use a COM-based application server (that is, not *TSoapDataModule*).

To create the parent remote data module, you must extend its *IAppServer* interface, adding properties that expose the interfaces of the child remote data modules. That is, for each child remote data module, add a property to the parent data module's interface whose value is the *IAppServer* interface for the child data module. The property getter should look something like the following:

```
function ParentRDM.Get_ChildRDM: IChildRDM;
begin
  if not Assigned(ChildRDMFactory) then
    ChildRDMFactory :=
      TComponentFactory.Create(ComServer, TChildRDM, Class_ChildRDM,
                               ciInternal, tmApartment);
```

```
  Result := ChildRDMFactory.CreateCOMObject(nil) as IChildRDM;
  Result.MainRDM := Self;
end;
```

For information about extending the parent remote data module's interface, see Extending the application server's interface (see page 1531).

Tip: You may also want to extend the interface for each child data module, exposing the parent data module's interface, or the interfaces of the other child data modules. This lets the various data modules in your application server communicate more freely with each other.

Once you have added properties that represent the child remote data modules to the main remote data module, client applications do not need to form separate connections to each remote data module on the application server. Instead, they share a single connection to the parent remote data module, which then dispatches messages to the "child" data modules. Because each client application uses the same connection for every remote data module, the remote data modules can share a single database connection, conserving resources. For information on how child applications share a single connection, see Connecting to an Application Server That Uses Multiple Data Modules (see page 1526).

See Also

Managing Transactions in Multi-tiered Applications (see page 1533)

Extending the Application Servers Interface (see page 1531)

Supporting Master/detail Relationships (see page 1539)

Supporting State Information in Remote Data Modules (see page 1540)

Setting Up the Remote Data Module (see page 1536)

3.2.3.3.44 Using SOAP Connections

SOAP is the protocol that underlies the built-in support for Web Service applications. SOAP marshals method calls using an XML encoding. SOAP connections use HTTP as a transport protocol.

SOAP connections have the advantage that they work in cross-platform applications because they are supported on both the Windows and Linux. Because SOAP connections use HTTP, they have the same advantages as Web connections: HTTP provides a lowest common denominator that you know is available on all clients, and clients can communicate with an application server that is protected by a "firewall." For more information about using SOAP to distribute applications, see Using Web Services. (see page 2291).

As with HTTP connections, you can't use callbacks when the connection is formed via SOAP.

See Also

Using DCOM Connections (see page 1546)

Using Socket Connections (see page 1548)

3.2.3.3.45 Using Socket Connections

TCP/IP Sockets let you create lightweight clients. For example, if you are writing a Web-based client application (see page 1551), you can't be sure that client systems support DCOM. Sockets provide a lowest common denominator that you know will be available for connecting to the application server. For more information about sockets, see Working with Sockets. (see page 2337)

Instead of instantiating the remote data module directly from the client (as happens with DCOM), sockets use a separate application on the server (ScktSrvr.exe), which accepts client requests and instantiates the remote data module using COM. The connection component on the client and ScktSrvr.exe on the server are responsible for marshaling *IAppServer* calls.

Note: ScktSrvr.exe can run as an NT service application. Register it with the Service manager by starting it using the -install command line option. You can unregister it using the -uninstall command line option.

Before you can use a socket connection, the application server must register its availability to clients using a socket connection. By default, all new remote data modules automatically register themselves by adding a call to EnableSocketTransport in the *UpdateRegistry* method. You can remove this call to prevent socket connections to your application server.

Note: Because older servers did not add this registration, you can disable the check for whether an application server is registered by unchecking the Connections->Registered Objects Only

menu item on ScktSrvr.exe. When using sockets, there is no protection on the server against client systems failing before they release a reference to interfaces on the application server. While this results in less message traffic than when using DCOM (which sends periodic keep-alive messages), this can result in an application server that can't release its resources because it is unaware that the client has gone away.

See Also

Using DCOM Connections (see page 1546)

Using SOAP Connections (see page 1548)

3.2.3.3.46 Using the Javascript Libraries

The HTML pages generated by the InternetExpress components and the Web items they contain make use of several javascript libraries that ship in the source/webmidas directory:

Javascript libraries

Library	Description
xmldom.js	This library is a DOM-compatible XML parser written in javascript. It allows parsers that do not support XML to use XML data packets. Note that this does not include support for XML Islands, which are supported by IE5 and later.
xmldb.js	This library defines data access classes that manage XML data packets and XML delta packets.
xmldisp.js	This library defines classes that associate the data access classes in xmldb with HTML controls in the HTML page.
xmlerrdisp.js	This library defines classes that can be used when reconciling update errors. These classes are not used by any of the built-in InternetExpress components, but are useful when writing a Reconcile producer.
xmlshow.js	This library includes functions to display formatted XML data packets and XML delta packets. This library is not used by any of the InternetExpress components, but is useful when debugging.

Once you have installed these libraries, you must set the *IncludePathURL* property of all InternetExpress page producers to indicate where they can be found.

It is possible to write your own HTML pages using the javascript classes provided in these libraries instead of using Web items to generate your Web pages. However, you must ensure that your code does not do anything illegal, as these classes include minimal error checking (so as to minimize the size of the generated Web pages).

See Also

Building an InternetExpress Application (see page 1521)

Granting Permission to Access and Launch the Application Server (see page 1532)

3.2.3.3.47 Using the Web Page Editor

The Web page editor lets you add Web items to your InternetExpress page producer and view the resulting HTML page. Display

the Web page editor by double-clicking on a InternetExpress page producer component.

Note: You must have Internet Explorer 4 or better installed to use the Web page editor.

The top of the Web page editor displays the Web items that generate the HTML document. These Web items are nested, where each type of Web item assembles the HTML generated by its subitems. Different types of items can contain different subitems. On the left, a tree view displays all of the Web items, indicating how they are nested. On the right, you can see the Web items included by the currently selected item. When you select a component in the top of the Web page editor, you can set its properties (see page 1536) using the **Object Inspector**.

Click the New Item button to add a subitem to the currently selected item. The Add Web Component dialog lists only those items that can be added to the currently selected item.

The InternetExpress page producer can contain one of two types of item, each of which generates an HTML form:

TDataForm, which generates an HTML form for displaying data and the controls that manipulate that data or submit updates.

Items you add to *TDataForm* display data in a multi-record grid (TDataGrid) or in a set of controls each of which represents a single field from a single record (TFieldGroup). In addition, you can add a set of buttons to navigate through data or post updates (TDataNavigator), or a button to apply updates back to the Web client (TApplyUpdatesButton). Each of these items contains subitems to represent individual fields or buttons. Finally, as with most Web items, you can add a layout grid (TLayoutGroup), that lets you customize the layout of any items it contains.

TQueryForm, which generates an HTML form for displaying or reading application-defined values. For example, you can use this form for displaying and submitting parameter values.

Items you add to *TQueryForm* display application-defined values(TQueryFieldGroup) or a set of buttons to submit or reset those values (TQueryButtons). Each of these items contains subitems to represent individual values or buttons. You can also add a layout grid to a query form, just as you can to a data form.

The bottom of the Web page editor displays the generated HTML code and lets you see what it looks like in a browser (Internet Explorer).

See Also

Customizing the InternetExpress Page Producer Template (see page 1529)

Setting Web Item Properties (see page 1536)

3.2.3.3.48 Using Web Connections

HTTP lets you create clients that can communicate with an application server that is protected by a firewall. HTTP messages provide controlled access to internal applications so that you can distribute your client applications safely and widely. Like socket connections, HTTP messages provide a lowest common denominator that you know will be available for connecting to the application server. For more information about HTTP messages, see Creating Internet Server Applications (see page 2251)

Instead of instantiating the remote data module directly from the client (as happens with DCOM), HTTP-based connections use a Web server application on the server (httpsrvr.dll) that accepts client requests and instantiates the remote data module using COM. Because of this, they are also called Web connections. The connection component on the client and httpsrvr.dll on the server are responsible for marshaling *IAppServer* calls.

Web connections can take advantage of the SSL security provided by wininet.dll (a library of Internet utilities that runs on the client system). Once you have configured the Web server on the server system to require authentication, you can specify the user name and password using the properties of the Web connection component.

As an additional security measure, the application server must register its availability to clients using a Web connection. By default, all new remote data modules automatically register themselves by adding a call to EnableWebTransport in the *UpdateRegistry* method. You can remove this call to prevent Web connections to your application server.

Web connections can take advantage of object pooling (see page 1535). This allows your server to create a limited pool of remote data module instances that are available for client requests. By pooling the remote data modules, your server does not consume the resources for the data module and its database connection except when they are needed.

Unlike most other connection components, you can't use callbacks when the connection is formed via HTTP.

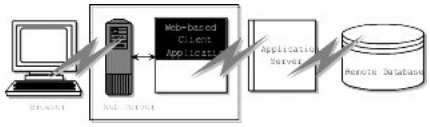
See Also

Using DCOM Connections (see page 1546)

Using SOAP Connections (see page 1548)

3.2.3.3.49 Writing Web-based Client Applications

If you want to create Web-based clients for your multi-tiered database application, you must replace the client tier with a special Web application that acts simultaneously as a client to an application server and as a Web server application that is installed with a Web server on the same machine. This architecture is illustrated in the following figure.



There are two approaches that you can take to build the Web application:

- You can combine the multi-tiered database architecture with an ActiveX form to distribute the client application as an ActiveX control (see page 1530). This allows any browser that supports ActiveX to run your client application as an in-process server.
- You can use XML data packets to build an InternetExpress application (see page 1522). This allows browsers that supports javascript to interact with your client application through html pages.

These two approaches are very different. Which one you choose depends on the following considerations:

- Each approach relies on a different technology (ActiveX vs. javascript and XML). Consider what systems your end users will use. The first approach requires a browser to support ActiveX (which limits clients to a Windows platform). The second approach requires a browser to support javascript and the DHTML capabilities introduced by Netscape 4 and Internet Explorer 4.
- ActiveX controls must be downloaded to the browser to act as an in-process server. As a result, the clients using an ActiveX approach require much more memory than the clients of an HTML-based application.
- The InternetExpress approach can be integrated with other HTML pages. An ActiveX client must run in a separate window.
- The InternetExpress approach uses standard HTTP, thereby avoiding any firewall issues that confront an ActiveX application.
- The ActiveX approach provides greater flexibility in how you program your application. You are not limited by the capabilities of the javascript libraries. The client datasets used in the ActiveX approach surface more features (such as filters, ranges, aggregation, optional parameters, delayed fetching of BLOBs or nested details, and so on) than the XML brokers used in the InternetExpress approach.

  Warning: Your Web client application may look and act differently when viewed from different browsers. Test your application with the browsers you expect your end-users to use.

3.2.3.4 Creating reports with Rave Reports

Topics

Name	Description
Rave Component Overview (see page 1552)	This section provides an overview of the Rave Reports components. For detailed component information, see the documentation listed in Getting more information (see page 1555).

Getting More Information (see page 1555)	Delphi includes the following Nevrona Designs documentation for Rave Reports. ***Rave Reports documentation***
Getting Started with Rave Reports (see page 1555)	You can use Rave Reports in VCL applications to generate reports from database and non-database data.
Rave Reports: Overview (see page 1556)	Rave Reports is a component-based visual report design tool that simplifies the process of adding reports to an application. You can use Rave Reports to create a variety of reports, from simple banded reports to more complex, highly customized reports. Report features include: • Word wrapped memos • Full graphics • Justification • Precise page positioning • Printer configuration • Font control • Print preview • Reuse of report content • PDF, HTML, RTF, and text report renditions
Rave Visual Designer (see page 1556)	To launch the Rave Visual Designer, do one of the following: • Choose **Tools** ▸ **Rave Designer**. • Double-click a TRvProject component on a form. • Right-click a TRvProject component on a form, and choose Rave Visual Designer. For a detailed information on using the Rave Visual Designer, use the Help menu or see the Rave Reports documentation listed in Getting more information (see page 1555).

3.2.3.4.1 Rave Component Overview

This section provides an overview of the Rave Reports components. For detailed component information, see the documentation listed in Getting more information (see page 1555).

VCL components

The VCL components for Rave Reports are non-visual components that you add to a form in your VCL application. They are available on the Rave category of the **Tool palette**. There are four categories of components: engine, render, data connection and Rave project.

Engine components

The Engine components are used to generate reports. Reports can be generated from a pre-defined visual definition (using the *Engine* property of TRvProject) or by making calls to the Rave code-based API library from within the OnPrint event. The engine components are:

1. TRvNDRWriter
2. TRvSystem

Render components

The Render components are used to convert an NDR file (Rave snapshot report file) or a stream generated from TRvNDRWriter to a variety of formats. Rendering can be done programmatically or added to the standard setup and preview dialogs of TRvSystem by dropping a render component on an active form or data module within your application. The render components

are:

1. TRvRenderPreview
2. TRvRenderPrinter
3. TRvRenderPDF
4. TRvRenderHTML
5. TRvRenderRTF
6. TRvRenderText

Data connection components

The Data Connection components provide the link between application data and the Direct Data Views in visually designed Rave reports. The data connection components are:

1. TRvCustomConnection
2. TRvDataSetConnection
3. TRvTableConnection
4. TRvQueryConnection

Rave project component

The TRvProject component interfaces with and executes visually designed Rave reports within an application. Normally a TRvSystem component would be assigned to the *Engine* property. The reporting project (.rav) should be specified in the *ProjectFile* property or loaded into the DFM using the *StoreRAV* property. Project parameters can be set using the SetParam method and reports can be executed using the ExecuteReport method.

Reporting components

The following components are available in the Rave Visual Designer.

Project components

The Project toolbar provides the essential building blocks for all reports. The project components are:

1. TRaveProjectManager
2. TRaveReport
3. TRavePage

Data objects

Data objects connect to data or control access to reports from the Rave Reporting Server. The **File ▸ New Data Object** menu command displays the Data Connections dialog box, which you can use to create each of the data objects. The data object components are:

1. TRaveDatabase
2. TRaveDriverDataView
3. TRaveDirectDataView
4. TRaveSimpleSecurity
5. TRaveLookupSecurity

Standard components

The Standard toolbar provides components that are frequently used when designing reports. The standard components are:

1. TRaveText
2. TRaveMemo
3. TRaveSection

4. TRaveBitmap
5. TRaveMetaFile
6. TRaveFontMaster
7. TRavePageNumInit

Drawing components

The Drawing toolbar provides components to create lines and shapes in a report. To color and style the components, use the Fills, Lines, and Colors toolbars. The drawing components are:

1. TRaveLine
2. TRaveHLine
3. TRaveVLine
4. TRaveSquare
5. TRaveRectangle
6. TRaveCircle
7. TRaveEllipse

Report components

The Report toolbar provides components that are used most often in data-aware reports. The report components are:

1. TRaveRegion
2. TRaveDataBand
3. TRaveBand
4. Band Style Edito
5. TRaveDataText
6. DataText Editor
7. TRaveDataMemo
8. TRaveCalcText
9. TRaveDataCycle
10. TRaveDataMirrorSection
11. TRaveCalcOp Component
12. TRaveCalcController
13. TRaveCalcTotal

Bar code components

The Bar Code toolbar provides different types of bar codes in a report. The bar code components are:

1. TRavePostNetBarCode
2. TRaveI2of5Bar Code
3. TRaveCode39BarCode
4. TRaveCode128BarCode
5. TRaveUPCBarCode
6. TRaveEANBarCode

3.2.3.4.2 Getting More Information

Delphi includes the following Nevrona Designs documentation for Rave Reports.

Rave Reports documentation

Title	Description
Rave Visual Designer Manual for Reference and Learning	Provides detailed information about using the Rave Visual Designer to create reports.
Rave Tutorial and Reference	Provides step-by-step instructions on using the Rave Reports components and includes a reference of classes, components, and units.
Rave Application Interface Technology Specification	Explains how to create custom Rave Reports components, property editors, component editors, project editors, and control the Rave environment.

These books are distributed as PDF files on the Delphi Companion Tools CD.

Most of the information in the PDF files is also available in the online Help. To display online Help for a Rave Reports component on a form, select the component and press `F1`. To display online Help for the Rave Visual Designer, use the Help menu.

3.2.3.4.3 Getting Started with Rave Reports

You can use Rave Reports in VCL applications to generate reports from database and non-database data.

To add a simple report to an existing database application

1. Open a database application in Delphi.
2. From the Rave category of the **Tool palette**, add the TRvDataSetConnection component to a form in the application.
3. In the **Object Inspector**, set the *DataSet* property to a dataset component that is already defined in your application.
4. Use the Rave Visual Designer:
5. From the Rave category of the **Tool palette**, add the Rave project component, TRvProject, to the form.
6. In the **Object Inspector**, set the *ProjectFile* property to the report project file (MyRave.rav) that you created in step 8 in using the Rave Visual Designer.
7. From the Standard category of the **Tool palette**, add the TButton component.
8. In the **Object Inspector**, click the Events tab and double-click the OnClick event.
9. Write an event handler that uses the ExecuteReport method to execute the Rave project component.

To design your report and create a report project file (.rav file) using the Rave Visual Designer

1. Choose **Tools** ▸ **Rave Designer** to launch the Rave Visual Designer.
2. Choose **File** ▸ **New Data Object** to display the Data Connections dialog box, and in the Data Object Type list, select Direct Data View and click Next.
3. In the Active Data Connections list, select RVDataSetConnection1 and click Finish. In the Project Tree on the left side of the Rave Visual Designer window, expand the Data View Dictionary node, then expand the newly created DataView1 node. Your application data fields are displayed under the DataView1 node.
4. Choose **Tools** ▸ **Report Wizards** ▸ **Simple Table** to display the Simple Table wizard, and select DataView1 and click Next.
5. Select two or three fields that you want to display in the report and click Next.
6. Follow the prompts on the subsequent wizard pages to set the order of the fields, margins, heading text, and fonts to be used in the report.

7. On the final wizard page, click Generate to complete the wizard and display the report in the Page Designer.
8. Choose **File** ▸ **Save** as to display the Save As dialog box. Navigate to the directory in which your Delphi application is located and save the Rave project file as MyRave.rav.
9. Minimize the Rave Visual Designer window and return to Delphi.

For a more information on using the Rave Visual Designer, use the Help menu or see the Rave Reports documentation listed in Getting more information (see page 1555).

3.2.3.4.4 Rave Reports: Overview

Rave Reports is a component-based visual report design tool that simplifies the process of adding reports to an application. You can use Rave Reports to create a variety of reports, from simple banded reports to more complex, highly customized reports. Report features include:

- Word wrapped memos
- Full graphics
- Justification
- Precise page positioning
- Printer configuration
- Font control
- Print preview
- Reuse of report content
- PDF, HTML, RTF, and text report renditions

3.2.3.4.5 Rave Visual Designer

To launch the Rave Visual Designer, do one of the following:

- Choose **Tools** ▸ **Rave Designer**.
- Double-click a TRvProject component on a form.
- Right-click a TRvProject component on a form, and choose Rave Visual Designer.

For a detailed information on using the Rave Visual Designer, use the Help menu or see the Rave Reports documentation listed in Getting more information (see page 1555).

3.2.3.5 Designing database applications

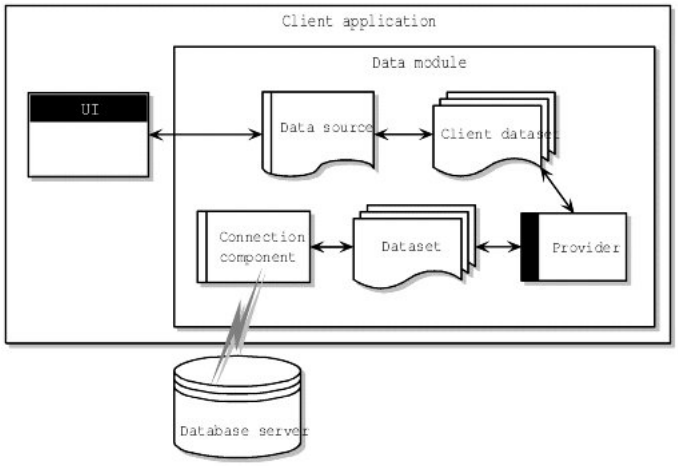
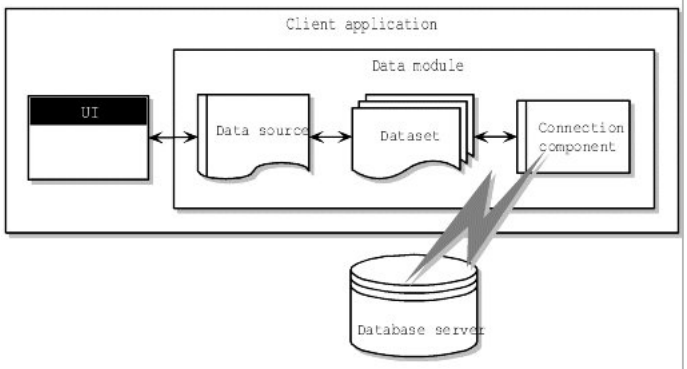
Topics

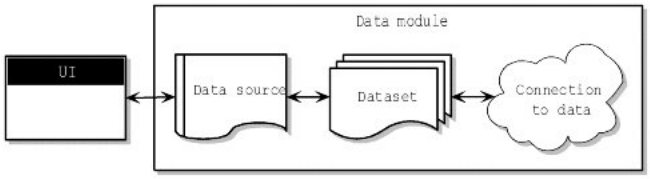
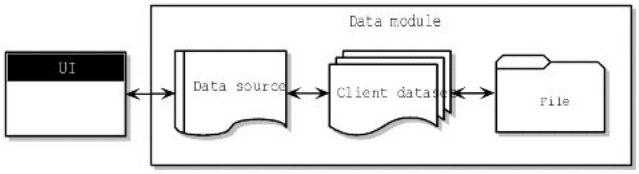
Name	Description
Database architecture (see page 1557)	
Designing Database Applications: Overview (see page 1566)	Database applications let users interact with information that is stored in databases. Databases provide structure for the information, and allow it to be shared among different applications. Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus. When designing a database application, you must understand how the data is structured. Based on that structure, you can then design a user interface to display data to the user and allow the user to enter new information or modify... more (see page 1566)
Designing the user interface (see page 1567)	

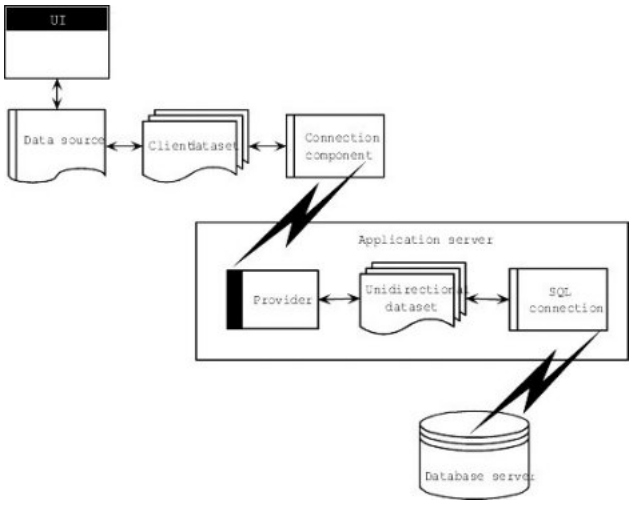
Using databases (see page 1568)	

3.2.3.5.1 Database architecture

Topics

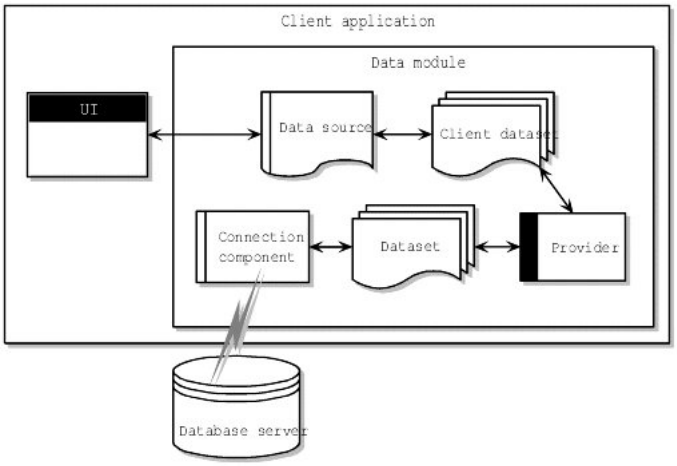
Name	Description
Connecting a Client Dataset to Another Dataset in the Same Application (see page 1559)	By using a provider component (see page 1819), you can connect *TClientDataSet* to another (source) dataset. The provider packages database information into transportable data packets (which can be used by client datasets) and applies updates received in delta packets (which client datasets create) back to a database server. The architecture for this is illustrated in the following figure.  This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database or a remote database server (see page 1571). The logic that manipulates database information is in the same application that implements the user interface, although isolated into... more (see page 1559)
Connecting Directly to a Database Server (see page 1560)	The most common database architecture is the one where the dataset uses a connection component to establish a connection to a database server. The dataset then fetches data directly from the server and posts edits directly to the server. This is illustrated in the following figure. Connecting directly to the database server  Each type of dataset uses its own type of connection component, which represents a single data access mechanism: • If the dataset is a BDE dataset such as *TTable*, *TQuery*, or *TStoredProc*, the connection component is a TDataBase*object. You connect the dataset to the database... more (see page 1560)*

Connecting to Another Dataset (see page 1561)	There are specialized client datasets that use the BDE or *dbExpress* to connect to a database server. These specialized client datasets are, in fact, composite components that include another dataset internally to access the data and an internal provider component to package the data from the source dataset and to apply updates back to the database server. These composite components require some additional overhead, but provide certain benefits: • Client datasets provide the most robust way to work with cached updates. By default, other types of datasets post edits directly to the database server. You can reduce network traffic by using... more (see page 1561)
Database Architecture (see page 1562)	Database applications are built from user interface elements, components that represent database information (datasets), and components that connect these to each other and to the source of the database information. How you organize these pieces is the architecture of your database application. While there are many distinct ways to organize the components in a database application, most follow the general scheme illustrated in the following figure: 
Using a Dedicated File on Disk (see page 1564)	The simplest form of database application you can write does not use a database server at all. Instead, it uses MyBase, the ability of client datasets to save themselves to a file and to load the data from a file. This architecture is illustrated in the following figure:  When using this file-based approach, your application writes changes to disk using the client dataset's SaveToFile method. *SaveToFile* takes one parameter, the name of the file which is created (or overwritten) containing the table. When you want to read a table previously written using the *SaveToFile* method, use the LoadFromFile method. *LoadFromFile... more (see page 1564)*

Using a Multi-Tiered Architecture (see page 1564)	When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a multi-tiered application. Multi-tiered applications have middle tiers between the client application and database server. The architecture for this is illustrated in the following figure.  The preceding figure represents three-tiered application. The logic that manipulates database information is on a separate system, or tier. This middle tier centralizes the logic that governs your database interactions so there is centralized control over data relationships. This allows different client applications to use the same data, while ensuring consistent data logic.... more (see page 1564)
Combining Approaches (see page 1565)	There is no reason why you can't combine two or more of the available architectures in a single application. In fact, some combinations can be extremely powerful. For example, you can combine the disk-based architecture described in Using a dedicated file on disk (see page 1564) with another approach such as those described in Connecting to another dataset (see page 1561) or Using a multi-tiered architecture (see page 1564). These combinations are easy because all models use a client dataset to represent the data that appears in the user interface. The result is called the briefcase model (or sometimes the disconnected model, or mobile computing). The briefcase model... more (see page 1565)

3.2.3.5.1.1 Connecting a Client Dataset to Another Dataset in the Same Application

By using a provider component (see page 1819), you can connect *TClientDataSet* to another (source) dataset. The provider packages database information into transportable data packets (which can be used by client datasets) and applies updates received in delta packets (which client datasets create) back to a database server. The architecture for this is illustrated in the following figure.



This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database or a remote database server (see page 1571). The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

To link the client dataset to the provider, set its ProviderName property to the name of the provider component. The provider must be in the same data module as the client dataset. To link the provider to the source dataset, set its DataSet property.

Once the client dataset is linked to the provider and the provider is linked to the source dataset, these components automatically handle all the details necessary for fetching, displaying, and navigating through the database records (assuming the source dataset is connected to a database). To apply user edits back to the database, you need only call the client dataset's ApplyUpdates method.

For more information on using a client dataset with a provider, see Using a client dataset with a provider (see page 1732).

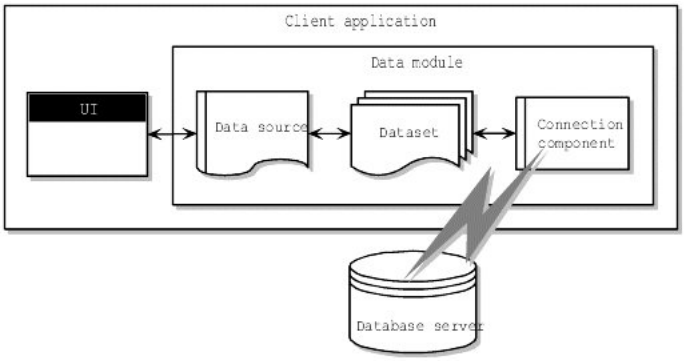
See Also

Using a multi-tiered architecture (see page 1564)

3.2.3.5.1.2 Connecting Directly to a Database Server

The most common database architecture is the one where the dataset uses a connection component to establish a connection to a database server. The dataset then fetches data directly from the server and posts edits directly to the server. This is illustrated in the following figure.

Connecting directly to the database server



Each type of dataset uses its own type of connection component, which represents a single data access mechanism:

- If the dataset is a BDE dataset such as *TTable*, *TQuery*, or *TStoredProc*, the connection component is a TDataBase*object.*

You connect the dataset to the database component by setting its Databaseproperty. You do not need to explicitly add a database component when using BDE dataset. If you set the dataset's DatabaseName property, a database component is created for you automatically at runtime.

- If the dataset is an ADO dataset such as *TADODataSet*, *TADOTable*, *TADOQuery*, or *TADOStoredProc*, the connection component is a TADOConnection*object. You connect the dataset to the ADO connection component by setting its Connectionproperty. As with BDE datasets, you do not need to explicitly add the connection component: instead you can set the dataset's ConnectionStringproperty.*
- If the dataset is a dbExpress dataset such as *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, or *TSQLStoredProc*, the connection component is a TSQLConnection object. You connect the dataset to the SQL connection component by setting its SQLConnection property. When using dbExpress datasets, you must explicitly add the connection component. Another difference between dbExpress datasets and the other datasets is that dbExpress datasets are always read-only and unidirectional: This means you can only navigate by iterating through the records in order, and you can't use the dataset methods that support editing.
- If the dataset is an InterBase Express dataset such as *TIBDataSet*, *TIBTable*, *TIBQuery*, or *TIBStoredProc*, the connection component is a TIBDatabase*object. You connect the dataset to the IB database component by setting its Database_Database">Databaseproperty. As with dbExpress datasets, you must explicitly add the connection component.*

In addition to the components listed above, you can use a specialized client dataset such as *TBDEClientDataSet, TSimpleDataSet*, or *TIBClientDataSet* with a database connection component. When using one of these client datasets, specify the appropriate type of connection component as the value of the *DBConnection* property.

Although each type of dataset uses a different connection component, they all perform many of the same tasks and surface many of the same properties, methods, and events. For more information on the commonalities among the various database connection components, see Connecting to databases (see page 1506)

This architecture represents either a single-tiered or two-tiered application, depending on whether the database server is a local database such or a remote database server (see page 1571). The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

Note: The connection components or drivers needed to create two-tiered applications are not available in all version of Delphi.

See Also

Using a Dedicated File on Disk (see page 1564)

Connecting to Another Dataset (see page 1561)

Combining Approaches (see page 1565)

Understanding Datasets (see page 1632)

3.2.3.5.1.3 Connecting to Another Dataset

There are specialized client datasets that use the BDE or *dbExpress* to connect to a database server. These specialized client datasets are, in fact, composite components that include another dataset internally to access the data and an internal provider component to package the data from the source dataset and to apply updates back to the database server. These composite components require some additional overhead, but provide certain benefits:

- Client datasets provide the most robust way to work with cached updates. By default, other types of datasets post edits directly to the database server. You can reduce network traffic by using a dataset that caches updates locally and applies them all later in a single transaction. For information on the advantages of using client datasets to cache updates, see Using a client dataset to cache updates (see page 1731).
- Client datasets can apply edits directly to a database server when the dataset is read-only. When using *dbExpress*, this is the only way to edit the data in the dataset (it is also the only way to navigate freely in the data when using *dbExpress*). Even when not using *dbExpress*, the results of some queries and all stored procedures are read-only. Using a client dataset provides a standard way to make such data editable.
- Because client datasets can work directly with dedicated files on disk, using a client dataset can be combined with a file-based model (see page 1565) to allow for a flexible "briefcase" application.

In addition to these specialized client datasets, there is a generic client dataset (*TClientDataSet*), which does not include an

internal dataset and dataset provider. Although *TClientDataSet* has no built-in database access mechanism, you can connect it to another, external, dataset from which it fetches data and to which it sends updates. Although this approach is a bit more complicated, there are times when it is preferable:

- Because the source dataset and dataset provider are external, you have more control over how they fetch data and apply updates. For example, the provider component surfaces a number of events that are not available when using a specialized client dataset to access data.
- When the source dataset is external, you can link it in a master/detail relationship with another dataset. An external provider automatically converts this arrangement into a single dataset with nested details. When the source dataset is internal, you can't create nested detail sets this way.
- Connecting a client dataset to an external dataset is an architecture that easily scales up to multiple tiers. Because the development process can get more involved and expensive as the number of tiers increases, you may want to start developing your application as a single-tiered or two-tiered application. As the amount of data, the number of users, and the number of different applications accessing the data grows, you may later need to scale up to a multi-tiered architecture. If you think you may eventually use a multi-tiered architecture, it can be worthwhile to start by using a client dataset with an external source dataset. This way, when you move the data access and manipulation logic to a middle tier, you protect your development investment because the code can be reused as your application grows.
- *TClientDataSet* can link to any source dataset. This means you can use custom datasets (third-party components) for which there is no corresponding specialized client dataset. Some versions of Delphi even include special provider components that connect a client dataset to an XML document rather than another dataset. (This works the same way as connecting a client dataset to another (source) dataset, except that the XML provider uses an XML document rather than a dataset. For information about these XML providers, see Using an XML document as the source (see page 1846) for a provider.)

There are two versions of the architecture that connects a client dataset to an external dataset:

- Connecting a client dataset to another dataset in the same application (see page 1559).
- Using a multi-tiered architecture (see page 1564).

See Also

Connecting Directly to a Database Server (see page 1560)

Using a Dedicated File on Disk (see page 1564)

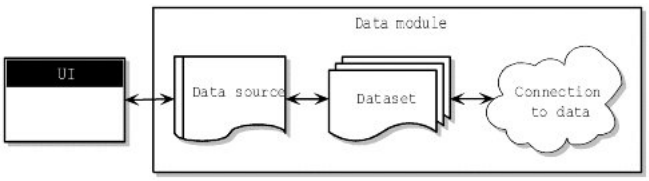
Combining Approaches (see page 1565)

Creating and Using a Client Dataset (see page 1740)

3.2.3.5.1.4 Database Architecture

Database applications are built from user interface elements, components that represent database information (datasets), and components that connect these to each other and to the source of the database information. How you organize these pieces is the architecture of your database application.

While there are many distinct ways to organize the components in a database application, most follow the general scheme illustrated in the following figure:



The user interface form

It is a good idea to isolate the user interface on a form that is completely separate from the rest of the application. This has several advantages. By isolating the user interface from the components that represent the database information itself, you introduce a greater flexibility into your design: Changes to the way you manage the database information do not require you to rewrite your user interface, and changes to the user interface do not require you to change the portion of your application that

works with the database. In addition, this type of isolation lets you develop common forms that you can share between multiple applications, thereby providing a consistent user interface. By storing links to well-designed forms in the Object Repository (see page 1916), you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms also makes it possible for you to develop corporate standards for application interfaces. For more information about creating the user interface for a database application, see Designing the user interface (see page 1567).

The data module

If you have isolated your user interface into its own form, you can use a data module (see page 1911) to house the components that represent database information (datasets), and the components that connect these datasets to the other parts of your application. Like the user interface forms, you can share data modules in the Object Repository so that they can be reused or shared between applications.

The data source

The first item in the data module is a data source (see page 1746). The data source acts as a conduit between the user interface and a dataset that represents information from a database. Several data-aware controls on a form can share a single data source, in which case the display in each control is synchronized so that as the user scrolls through records, the corresponding value in the fields for the current record is displayed in each control.

The dataset

The heart of your database application is the dataset. This component represents a set of records from the underlying database. These records can be the data from a single database table, a subset of the fields or records in a table, or information from more than one table joined into a single view. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. When the underlying database changes, you might need to alter the way the dataset component specifies the data it contains, but the rest of your application can continue to work without alteration. For more information on the common properties and methods of datasets, see Understanding datasets (see page 1632)

The data connection

Different types of datasets use different mechanisms for connecting to the underlying database information. These different mechanisms, in turn, make up the major differences in the architecture of the database applications you can build. There are four basic mechanisms for connecting to the data:

- Connecting directly to a database server (see page 1560). Most datasets use a descendant of *TCustomConnection* to represent the connection to a database server.
- Using a dedicated file on disk (see page 1564). Client datasets support the ability to work with a dedicated file on disk. No separate connection component is needed when working with a dedicated file because the client dataset itself knows how to read from and write to the file.
- Connecting to another dataset (see page 1561). Client datasets can work with data provided by another dataset. A *TDataSetProvider* component serves as an intermediary between the client dataset and its source dataset. This dataset provider can reside in the same data module as the client dataset, or it can be part of an application server running on another machine. If the provider is part of an application server, you also need a special descendant of *TCustomConnection* to represent the connection to the application server.
- Obtaining data from an RDS DataSpace object. ADO datasets can use a TRDSConnection component to marshal data in multi-tier database applications that are built using ADO-based application servers.

Sometimes, these mechanisms can be combined in a single application (see page 1565).

See Also

Designing the User Interface (see page 1567)

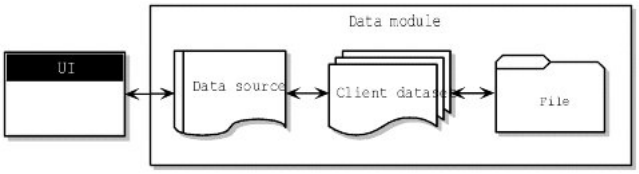
Using the Borland Database Engine (see page 1643)

Using Databases (see page 1572)

Types of Databases (see page 1571)

3.2.3.5.1.5 Using a Dedicated File on Disk

The simplest form of database application you can write does not use a database server at all. Instead, it uses MyBase, the ability of client datasets to save themselves to a file and to load the data from a file. This architecture is illustrated in the following figure:



When using this file-based approach, your application writes changes to disk using the client dataset's SaveToFile method. *SaveToFile* takes one parameter, the name of the file which is created (or overwritten) containing the table. When you want to read a table previously written using the *SaveToFile* method, use the LoadFromFile method. *LoadFromFile* also takes one parameter, the name of the file containing the table.

If you always load to and save from the same file, you can use the FileName property instead of the *SaveToFile* and *LoadFromFile* methods. When *FileName* is set to a valid file name, the data is automatically loaded from the file when the client dataset is opened and saved to the file when the client dataset is closed.

This simple file-based architecture is a single-tiered application. The logic that manipulates database information is in the same application that implements the user interface, although isolated into a data module.

The file-based approach has the benefit of simplicity. There is no database server to install, configure, or deploy (If you do not statically link in midaslib.dcu, the client dataset does require midas.dll). There is no need for site licenses or database administration.

In addition, some versions of Delphi let you convert between arbitrary XML documents and the data packets that are used by a client dataset. Thus, the file-based approach can be used to work with XML documents as well as dedicated datasets. For information about converting between XML documents and client dataset data packets, see Using XML in database applications (see page 1847)

The file-based approach offers no support for multiple users. The dataset should be dedicated entirely to the application. Data is saved to files on disk, and loaded at a later time, but there is no built-in protection to prevent multiple users from overwriting each other's data files.

For more information about using a client dataset with data stored on disk, see Using a client dataset with file-based data (see page 1733).

See Also

Connecting Directly to a Database Server (see page 1560)

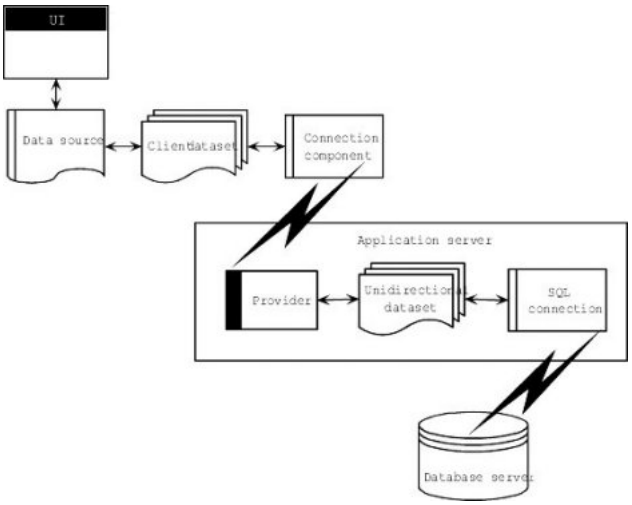
Connecting to Another Dataset (see page 1561)

Combining Approaches (see page 1565)

Creating and Using a Client Dataset (see page 1740)

3.2.3.5.1.6 Using a Multi-Tiered Architecture

When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a multi-tiered application. Multi-tiered applications have middle tiers between the client application and database server. The architecture for this is illustrated in the following figure.



The preceding figure represents three-tiered application. The logic that manipulates database information is on a separate system, or tier. This middle tier centralizes the logic that governs your database interactions so there is centralized control over data relationships. This allows different client applications to use the same data, while ensuring consistent data logic. It also allows for smaller client applications because much of the processing is off-loaded onto the middle tier. These smaller client applications are easier to install, configure, and maintain. Multi-tiered applications can also improve performance by spreading data-processing over several systems.

The multi-tiered architecture is very similar to the model described in Connecting a client dataset to another dataset in the same application (see page 1559). It differs mainly in that source dataset that connects to the database server and the provider that acts as an intermediary between that source dataset and the client dataset have both moved to a separate application. That separate application is called the application server (or sometimes the "remote data broker").

Because the provider has moved to a separate application, the client dataset can no longer connect to the source dataset by simply setting its ProviderName property. In addition, it must use some type of connection component to locate and connect to the application server.

There are several types of connection components that can connect a client dataset to an application server. They are all descendants of TCustomRemoteServer, and differ primarily in the communication protocol they use (TCP/IP, HTTP, DCOM, or SOAP). Link the client dataset to its connection component by setting the RemoteServer property.

The connection component establishes a connection to the application server and returns an interface that the client dataset uses to call the provider specified by its *ProviderName* property. Each time the client dataset calls the application server, it passes the value of *ProviderName*, and the application server forwards the call to the provider.

For more information about connecting a client dataset to an application server, see Creating multi-tiered applications (see page 1518)

See Also

Connecting a Client Dataset to Another Dataset in the Same Application (see page 1559)

Using a client dataset with a data provider (see page 1732)

3.2.3.5.1.7 Combining Approaches

There is no reason why you can't combine two or more of the available architectures in a single application. In fact, some combinations can be extremely powerful.

For example, you can combine the disk-based architecture described in Using a dedicated file on disk (see page 1564) with another approach such as those described in Connecting to another dataset (see page 1561) or Using a multi-tiered

architecture (see page 1564). These combinations are easy because all models use a client dataset to represent the data that appears in the user interface. The result is called the briefcase model (or sometimes the disconnected model, or mobile computing).

The briefcase model is useful in a situation such as the following: An onsite company database contains customer contact data that sales representatives can use and update in the field. While onsite, sales representatives download information from the database. Later, they work with it on their laptops as they fly across the country, and even update records at existing or new customer sites. When the sales representatives return onsite, they upload their data changes to the company database for everyone to use.

When operating on site, the client dataset in a briefcase model application fetches its data from a provider. The client dataset is therefore connected to the database server and can, through the provider, fetch server data and send updates back to the server. Before disconnecting from the provider, the client dataset saves its snapshot of the information to a file on disk. While offsite, the client dataset loads its data from the file, and saves any changes back to that file. Finally, back onsite, the client dataset reconnects to the provider so that it can apply its updates to the database server or refresh its snapshot of the data.

See Also

Connecting Directly to a Database Server (see page 1560)

Using a Dedicated File on Disk (see page 1564)

Connecting to Another Dataset (see page 1561)

3.2.3.5.2 Designing Database Applications: Overview

Database applications let users interact with information that is stored in databases. Databases provide structure for the information, and allow it to be shared among different applications.

Delphi provides support for relational database applications. Relational databases organize information into tables, which contain rows (records) and columns (fields). These tables can be manipulated by simple operations known as the relational calculus.

When designing a database application, you must understand how the data is structured. Based on that structure, you can then design a user interface to display data to the user and allow the user to enter new information or modify existing data.

The following topics introduce common considerations when designing a database application:

- Using Databases (see page 1572)
- Database Architecture (see page 1562)
- Designing the User Interface (see page 1567)

See Also

Using Database Information in Responses (see page 2266)

Creating Applications (see page 1883)

3.2.3.5.3 Designing the user interface

Topics

Name	Description
Analyzing Data (see page 1567)	Some database applications do not present database information directly to the user. Instead, they analyze and summarize information from databases so that users can draw conclusions from the data. The TDBChart component on the Data Controls category of the **Tool Palette** lets you present database information in a graphical format that enables users to quickly grasp the import of database information. In addition, some versions of Delphi include a Decision Cube category on the **Tool Palette**. It contains six components that let you perform data analysis and cross-tabulations on data when building decision support applications. For more information... more (see page 1567)
Designing the User Interface (see page 1567)	The Data Controls category of the **Tool Palette** provides a set of data-aware controls that represent data from fields in a database record, and can permit users to edit that data and post changes back to the database. Using data-aware controls, you can build your database application's user interface (UI) so that information is visible and accessible to users. For more information on data-aware controls see Using data controls (see page 1778). In addition to the basic data controls, you may also want to introduce other elements into your user interface: • You may want your application to analyze the data (see page 1567) contained... more (see page 1567)
Writing Reports (see page 1568)	If you want to let your users print database information from the datasets in your application, you can use Rave Reports, as described in Creating reports with Rave Reports. (see page 1556)

3.2.3.5.3.1 Analyzing Data

Some database applications do not present database information directly to the user. Instead, they analyze and summarize information from databases so that users can draw conclusions from the data.

The TDBChart component on the Data Controls category of the **Tool Palette** lets you present database information in a graphical format that enables users to quickly grasp the import of database information.

In addition, some versions of Delphi include a Decision Cube category on the **Tool Palette**. It contains six components that let you perform data analysis and cross-tabulations on data when building decision support applications. For more information about using the Decision Cube components, see Using decision support components (see page 1800)

If you want to build your own components that display data summaries based on various grouping criteria, you can use maintained aggregates (see page 1735) with a client dataset.

See Also

Writing Reports (see page 1568)

Using Data Controls (see page 1778)

3.2.3.5.3.2 Designing the User Interface

The Data Controls category of the **Tool Palette** provides a set of data-aware controls that represent data from fields in a database record, and can permit users to edit that data and post changes back to the database. Using data-aware controls, you can build your database application's user interface (UI) so that information is visible and accessible to users. For more information on data-aware controls see Using data controls (see page 1778).

In addition to the basic data controls, you may also want to introduce other elements into your user interface:

- You may want your application to analyze the data (see page 1567) contained in a database. Applications that analyze data do more than just display the data in a database, they also summarize the information in useful formats to help users grasp

the impact of that data.

- You may want to print reports (see page 1568) that provide a hard copy of the information displayed in your user interface.
- You may want to create a user interface that can be viewed from Web browsers. The simplest Web-based database applications are described in Using database information in responses (see page 2266). In addition, you can combine the Web-based approach with the multi-tiered architecture, as described in Writing Web-based client applications. (see page 1551)

See Also

Using Databases (see page 1572)

Database Architecture (see page 1562)

Using Data Controls (see page 1778)

Designing the User Interface

3.2.3.5.3.3 Writing Reports

If you want to let your users print database information from the datasets in your application, you can use Rave Reports, as described in Creating reports with Rave Reports. (see page 1556)

See Also

Using Data Controls (see page 1778)

Analyzing Data (see page 1567)

3.2.3.5.4 Using databases

Topics

Name	Description
Database Security (see page 1569)	Databases often contain sensitive information. Different databases provide security schemes for protecting that information. Some databases, such as Paradox and dBASE, only provide security at the table or field level. When users try to access protected tables, they are required to provide a password. Once users have been authenticated, they can see only those fields (columns) for which they have permission. Most SQL servers require a password and user name to use the database server at all. Once the user has logged in to the database, that username and password determine which tables can be used. For information on providing... more (see page 1569)
Referential Integrity, Stored Procedures, and Triggers (see page 1570)	All relational databases have certain features in common that allow applications to store and manipulate data. In addition, databases often provide other, database-specific, features that can prove useful for ensuring consistent relationships between the tables in a database. These include • **Referential integrity.** Referential integrity provides a mechanism to prevent master/detail relationships between tables from being broken. When the user attempts to delete a field in a master table which would result in orphaned detail records, referential integrity rules prevent the deletion or automatically delete the orphaned detail records. • **Stored procedures.** Stored procedures are sets of SQL statements that are named... more (see page 1570)

Transactions (see page 1570)	A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If any of the actions in the group fails, then all actions are rolled back (undone). Transactions ensure that • All updates in a single transaction are either committed or aborted and rolled back to their previous state. This is referred to as **atomicity**. • A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**. • Concurrent transactions do not see each other's... more (see page 1570)
Types of Databases (see page 1571)	Relational database servers vary in the way they store information and in the way they allow multiple users to access that information simultaneously. Delphi provides support for two types of relational database server: • **Remote database servers** reside on a separate machine. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers. Although remote database servers vary in the way they store information, they provide a common logical interface to clients. This common interface is Structured Query Language (SQL). Because you access them using SQL, they are sometimes called... more (see page 1571)
Using Databases (see page 1572)	Delphi includes many components for accessing databases and representing the information they contain. They are grouped according to the data access mechanism: • The BDE page of the Component palette contains components that use the Borland Database Engine (BDE). The BDE defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables. However, it is also the most complicated mechanism to deploy. For more information about using the... more (see page 1572)

3.2.3.5.4.1 Database Security

Databases often contain sensitive information. Different databases provide security schemes for protecting that information. Some databases, such as Paradox and dBASE, only provide security at the table or field level. When users try to access protected tables, they are required to provide a password. Once users have been authenticated, they can see only those fields (columns) for which they have permission.

Most SQL servers require a password and user name to use the database server at all. Once the user has logged in to the database, that username and password determine which tables can be used. For information on providing passwords to SQL servers, see Controlling server login (see page 1507).

When designing database applications, you must consider what type of authentication is required by your database server. Often, applications are designed to hide the explicit database login from users, who need only log in to the application itself. If you do not want to require your users to provide a database password, you must either use a database that does not require one or you must provide the password and username to the server programmatically. When providing the password programmatically, care must be taken that security can't be breached by reading the password from the application.

If you require your user to supply a password, you must consider when the password is required. If you are using a local database but intend to scale up to a larger SQL server later, you may want to prompt for the password at the point when you will

eventually log in to the SQL database, rather than when opening individual tables.

If your application requires multiple passwords because you must log in to several protected systems or databases, you can have your users provide a single master password that is used to access a table of passwords required by the protected systems. The application then supplies passwords programmatically, without requiring the user to provide multiple passwords.

In multi-tiered applications, you may want to use a different security model altogether. You can use HTTPs or COM+ to control access to middle tiers, and let the middle tiers handle all details of logging into database servers.

See Also

Types of Databases (see page 1571)

Transactions (see page 1570)

The Data Dictionary (see page 1680)

Referential Integrity (see page 1570)

3.2.3.5.4.2 Referential Integrity, Stored Procedures, and Triggers

All relational databases have certain features in common that allow applications to store and manipulate data. In addition, databases often provide other, database-specific, features that can prove useful for ensuring consistent relationships between the tables in a database. These include

- **Referential integrity.** Referential integrity provides a mechanism to prevent master/detail relationships between tables from being broken. When the user attempts to delete a field in a master table which would result in orphaned detail records, referential integrity rules prevent the deletion or automatically delete the orphaned detail records.
- **Stored procedures.** Stored procedures are sets of SQL statements that are named and stored on an SQL server. Stored procedures usually perform common database-related tasks on the server, and sometimes return sets of records (datasets).
- **Triggers.** Triggers are sets of SQL statements that are automatically executed in response to a particular command.

See Also

Types of Databases (see page 1571)

Database Security (see page 1569)

Transactions (see page 1570)

The Data Dictionary (see page 1680)

3.2.3.5.4.3 Transactions

A transaction is a group of actions that must all be carried out successfully on one or more tables in a database before they are committed (made permanent). If any of the actions in the group fails, then all actions are rolled back (undone).

Transactions ensure that

- All updates in a single transaction are either committed or aborted and rolled back to their previous state. This is referred to as **atomicity**.
- A transaction is a correct transformation of the system state, preserving the state invariants. This is referred to as **consistency**.
- Concurrent transactions do not see each other's partial or uncommitted results, which might create inconsistencies in the application state. This is referred to as **isolation**.
- Committed updates to records survive failures, including communication failures, process failures, and server system failures. This is referred to as **durability**.

Thus, transactions protect against hardware failures that occur in the middle of a database command or set of commands. Transactional logging allows you to recover the durable state after disk media failures. Transactions also form the basis of

multi-user concurrency control on SQL servers. When each user interacts with the database only through transactions, one user's commands can't disrupt the unity of another user's transaction. Instead, the SQL server schedules incoming transactions, which either succeed as a whole or fail as a whole.

Transaction support is not part of most local databases, although it is provided by local InterBase. In addition, the BDE drivers provide limited transaction support for some local databases. Database transaction support is provided by the component that represents the database connection. For details on managing transactions using a database connection component, see Managing transactions (see page 1498).

In multi-tiered applications, you can create transactions that include actions other than database operations or that span multiple databases. For details on using transactions in multi-tiered applications, see Managing transactions in multi-tiered applications (see page 1533).

See Also

Types of Databases (see page 1571)

Database Security (see page 1569)

The Data Dictionary (see page 1680)

Referential Integrity (see page 1570)

3.2.3.5.4.4 Types of Databases

Relational database servers vary in the way they store information and in the way they allow multiple users to access that information simultaneously. Delphi provides support for two types of relational database server:

- **Remote database servers** reside on a separate machine. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers. Although remote database servers vary in the way they store information, they provide a common logical interface to clients. This common interface is Structured Query Language (SQL). Because you access them using SQL, they are sometimes called SQL servers. (Another name is Remote Database Management system, or RDBMS.) In addition to the common commands that make up SQL, most remote database servers support a unique "dialect" of SQL. Examples of SQL servers include InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2.
- **Local databases** reside on your local drive or on a local area network. They often have proprietary APIs for accessing the data. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases. Examples of local databases include Paradox, dBASE, FoxPro, and Access.

Applications that use local databases are called **single-tiered applications** because the application and the database share a single file system. Applications that use remote database servers are called **two-tiered applications** or **multi-tiered** applications because the application and the database operate on independent systems (or tiers).

Choosing the type of database to use depends on several factors. For example, your data may already be stored in an existing database. If you are creating the database tables your application uses, you may want to consider the following questions:

- How many users will be sharing these tables? Remote database servers are designed for access by several users at the same time. They provide support for multiple users through a mechanism called transactions (see page 1570). Some local databases (such as Local InterBase) also provide transaction support, but many only provide file-based locking mechanisms, and some (such as client dataset files) provide no multi-user support at all.
- How much data will the tables hold? Remote database servers can hold more data than local databases. Some remote database servers are designed for warehousing large quantities of data while others are optimized for other criteria (such as fast updates).
- What type of performance (speed) do you require from the database? Local databases are usually faster than remote database servers because they reside on the same system as the database application. Different remote database servers are optimized to support different types of operations, so you may want to consider performance when choosing a remote database server.
- What type of support will be available for database administration? Local databases require less support than remote database servers. Typically, they are less expensive to operate because they do not require separately installed servers or expensive site licenses.

See Also

Database Security (see page 1569)

Transactions (see page 1570)

Referential Integrity (see page 1570)

3.2.3.5.4.5 Using Databases

Delphi includes many components for accessing databases and representing the information they contain. They are grouped according to the data access mechanism:

- The BDE page of the Component palette contains components that use the Borland Database Engine (BDE). The BDE defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables. However, it is also the most complicated mechanism to deploy. For more information about using the BDE components, see Using the Borland Database Engine (see page 1643).
- The ADO page (see page 1494) of the Component palette contains components that use ActiveX Data Objects (ADO) to access database information through OLEDB. ADO is a Microsoft Standard. There is a broad range of ADO drivers available for connecting to different database servers. Using ADO-based components lets you integrate your application into an ADO-based environment (for example, making use of ADO-based application servers). For more information about using the ADO components, see Working with ADO Components (see page 1494)
- The dbExpress page (see page 16) of the Component palette contains components that use dbExpress to access database information. dbExpress is a lightweight set of drivers that provide the fastest access to database information. However, dbExpress database components also support the narrowest range of data manipulation functions. For more information about using the dbExpress components, see Using unidirectional datasets (see page 1823)
- The InterBase page of the Component palette contains components that access InterBase databases directly, without going through a separate engine layer.
- The Data Access page of the Component palette contains components that can be used with any data access mechanism. This page includes *TClientDataset*, which can work with data stored on disk or, using the *TDataSetProvider* component also on this page, with components from one of the other groups. For more information about using client datasets, see Using client datasets (see page 1740) For more information about *TDataSetProvider*, see Using provider components (see page 1819)

  Note: Different versions of Delphi include different drivers for accessing database servers using the BDE, ADO, or dbExpress.

  When designing a database application, you must decide which set of components to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

  In addition to choosing a data access mechanism, you must choose a database server. There are different types of databases (see page 1571) and you will want to consider the advantages and disadvantages of each type before settling on a particular database server.

  All types of databases contain tables which store information. In addition, most (but not all) servers support additional features such as

- Database security (see page 1569)
- Transactions (see page 1570)
- Referential integrity (see page 1570)

See Also

Designing the User Interface (see page 1567)

Using the Borland Database Engine (see page 1643)

Understanding Datasets (see page 1632)

Database Architecture (see page 1562)

Connecting to Databases (see page 1506)

3.2.3.6 Understanding datasets

Topics

Name	Description
Creating an Index with IndexFieldNames (see page 1581)	If there is no defined index that implements the sort order you want, you can create a pseudo-index using the *IndexFieldNames* property. **Note:** *IndexName and IndexFieldNames are mutually exclusive. Setting one property clears values set for the other. For information about IndexName, see Specifying an index with IndexName (see page 1630).* *The value of IndexFieldNames is a string. To specify a sort order, list each column name to use in the order it should be used, and delimit the names with semicolons. Sorting is by ascending order only. Case-sensitivity of the sort depends on the capabilities of your server. See your... more (see page 1581)*
Supplying Parameters at Design Time (see page 1582)	At design time, you can specify parameter values using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the **Object Inspector**. If the SQL statement does not contain any parameters, no objects are listed in the collection editor. **Note:** The parameter collection editor is the same collection editor that appears for other collection properties. Because the editor is shared with other properties, its right-click context menu contains the Add and Delete commands. However, they are never enabled for query parameters. The only way to add or... more (see page 1582)
Supplying Parameters at Runtime (see page 1583)	To create parameters at runtime, you can use the • *ParamByName* method to assign values to a parameter based on its name (not available for *TADOQuery*) • *Params* or *Parameters* property to assign values to a parameter based on the parameter's ordinal position within the SQL statement. • *Params.ParamValues* or *Parameters.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set. The following code uses *ParamByName* to assign the text of an edit box to the :Capital parameter:
Adding New Records (see page 1583)	A dataset must be in *dsInsert* mode before an application can add new records. In code, you can use the Insert or Append methods to put a dataset into *dsInsert* mode if the read-only CanModify property for the dataset is *True*. When a dataset transitions to *dsInsert* mode, it first receives a *BeforeInsert* event. After the transition to insert mode is successfully completed, the dataset receives first an *OnNewRecord* event hand then an *AfterInsert* event. You can use these events, for example, to provide initial values to newly inserted records:
Applying or Canceling a Range (see page 1585)	When you call *SetRangeStart* or *EditRangeStart* to specify the start of a range, or *SetRangeEnd* or *EditRangeEnd* to specify the end of a range, the dataset enters the *dsSetKey* state. It stays in that state until you apply or cancel the range.

Calculating Fields (see page 1585)	Using the Fields editor, you can define calculated fields (see page 1872) for your datasets. When a dataset contains calculated fields, you provide the code to calculate those field's values in an OnCalcFields event handler. The AutoCalcFields property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, *OnCalcFields* is called when • A dataset is opened. • The dataset enters edit mode. • A record is retrieved from the database. • Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified. If *AutoCalcFields* is *False*, then *OnCalcFields* is not... more (see page 1585)
Canceling Changes (see page 1586)	An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called Post. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the Cancel method for the dataset. A call to Cancel always returns a dataset to *dsBrowse* state. If the dataset was in *dsEdit* or *dsInsert* mode when your application called *Cancel*, it receives *BeforeCancel* and *AfterCancel* events before and after the current... more (see page 1586)
Controlling Read/Write Access to Tables (see page 1587)	By default when a table type dataset is opened, it requests read and write access for the underlying database table. Depending on the characteristics of the underlying database table, the requested write privilege may not be granted (for example, when you request write access to an SQL table on a remote server and the server restricts the table's access to read only). **Note:** This is not true for *TClientDataSet*, which determines whether users can edit data from information that the dataset provider supplies with data packets. It is also not true for *TSQLTable*, which is a unidirectional dataset,... more (see page 1587)
Creating and Deleting Tables (see page 1587)	Some table type datasets let you create and delete the underlying tables at design time or at runtime. Typically, database tables are created and deleted by a database administrator. However, it can be handy during application development and testing to create and destroy database tables that your application can use.
Creating Filters (see page 1590)	There are two ways to create a filter for a dataset: • Set the Filter property (see page 1608). Filter is especially useful for creating and applying filters at runtime. • Write an OnFilterRecord event handler (see page 1635) for simple or complex filter conditions. With OnFilterRecord, you specify filter conditions at design time. Unlike the *Filter* property, which is restricted to a single string containing filter logic, an *OnFilterRecord* event can take advantage of branching and looping logic to create complex, multi-level filter conditions. The main advantage to creating filters using the *Filter* property is that your application can create, change, and apply filters dynamically, (for... more (see page 1590)

Creating Master/detail Relationships (see page 1590)	Table type datasets can be linked into master/detail relationships. When you set up a master/detail relationship, you link two datasets so that all the records of one (the detail) always correspond to the single current record in the other (the master). Table type datasets support master/detail relationships in two very distinct ways: • All table type datasets can act as the detail of another dataset by linking cursors. This process is described in Making the table a detail of another dataset (see page 1598). • *TTable*, *TSQLTable*, and all client datasets can act as the master in a master/detail relationship that uses... more (see page 1590)
Deleting Records (see page 1591)	Use the Delete method to delete the current record in an active dataset. When the *Delete* method is called, • The dataset receives a *BeforeDelete* event. • The dataset attempts to delete the current record. • The dataset returns to the *dsBrowse* state. • The dataset receives an *AfterDelete* event. If want to prevent the deletion in the *BeforeDelete* event handler, you can call the global *Abort* procedure:
Editing Records (see page 1591)	A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit method to put a dataset into dsEdit mode if the read-only CanModify property for the dataset is True.* *When a dataset transitions to dsEdit mode, it first receives a BeforeEdit event. After the transition to edit mode is successfully completed, the dataset receives an AfterEdit event. Typically, these events are used for updating the user interface to indicate the current state of the dataset. If the dataset can't be put into edit mode for some reason, an... more (see page 1591)*
Emptying Tables (see page 1592)	Many table type datasets supply a single method that lets you delete all rows of data in the table.
Enabling and Disabling Filtering (see page 1593)	
Establishing Master/detail Relationships Using Parameters (see page 1593)	To set up a master/detail relationship where the detail set is a query-type dataset, you must specify a query that uses parameters. These parameters refer to current field values on the master dataset. Because the current field values on the master dataset change dynamically at runtime, you must rebind the detail set's parameters every time the master record changes. Although you could write code to do this using an event handler, all query-type datasets except *TIBQuery* provide an easier mechanism using the DataSource property. If parameter values for a parameterized query are not bound at design time or specified at... more (see page 1593)
Executing a Search with Find Methods (see page 1594)	The *Find* methods do the same thing as the Goto methods (see page 1595), except that you do not need to explicitly put the dataset in *dsSetKey* state to specify the key field values on which to search.
Executing a Search with Goto Methods (see page 1595)	
Executing Queries That Don't Return a Result Set (see page 1596)	When a query returns a set of records (such as a SELECT query), you execute the query the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method. However, often SQL commands do not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). For all query-type datasets, you can execute a query that does not return a result set by calling *ExecSQL... more (see page 1596)*
Executing Stored Procedures That Don't Return a Result Set (see page 1596)	When a stored procedure returns a cursor, you execute it the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method. However, often stored procedures do not return any data, or only return results in output parameters. You can execute a stored procedure that does not return a result set by calling *ExecProc*. After executing the stored procedure, you can use the *ParamByName* method to read the value of the result parameter or of any output parameters:

Fetching Multiple Result Sets (see page 1597)	Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. If you are using *TSQLStoredProc* or *TADOStoredProc*, you can access the other sets of records by calling the *NextRecordSet* method:
Limiting Records with Ranges (see page 1597)	You can temporarily view and edit a subset of data for any dataset by using filters (see page 1631). Some table type datasets support an additional way to access a subset of available records, called ranges. Ranges only apply to *TTable* and to client datasets. Despite their similarities, ranges and filters have different uses. The following topics discuss the differences between ranges and filters and how to use ranges: • Understanding the differences between ranges and filters (see page 1615) • Specifying ranges (see page 1610) • Modifying a range (see page 1601) • Applying or canceling a range (see page 1585)
Making the Table a Detail of Another Dataset (see page 1598)	A table type dataset's *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two datasets. The *MasterSource* property is used to specify a data source from which the table gets data from the master table. This data source can be linked to any type of dataset. For instance, by specifying a query's data source in this property, you can link a client dataset as the detail of the query, so that the client dataset tracks events occurring in the query. The dataset is linked to the master table based on its current index. Before you specify the... more (see page 1598)
Marking and Returning to Records (see page 1599)	In addition to moving from record to record in a dataset (or moving from one record to another by a specific number of records), it is often also useful to mark a particular location in a dataset so that you can return to it quickly when desired. TDataSet introduces a bookmarking feature that consists of a Bookmark property and five bookmark methods. *TDataSet* implements **virtual** bookmark methods. While these methods ensure that any dataset object derived from *TDataSet* returns a value if a bookmark method is called, the return values are merely defaults that do not keep track of the... more (see page 1599)
Modifying a Range (see page 1601)	Two functions enable you to modify the existing boundary conditions for a range: *EditRangeStart*, for changing the starting values for a range; and *EditRangeEnd*, for changing the ending values for the range.
Modifying Entire Records (see page 1601)	On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record. In code, however, you can use the following methods that work with entire record structures provided that the structure of the database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records: ***Methods that work with entire records***
Navigating Records in a Filtered Dataset (see page 1603)	There are four dataset methods that navigate among records in a filtered dataset. The following table lists these methods and describes what they do: ***Filtered dataset navigational methods***
Obtaining Information About Indexes (see page 1603)	Your application can obtain information about server-defined indexes from all table type datasets. To obtain a list of available indexes for the dataset, call the *GetIndexNames* method. *GetIndexNames* fills a string list with valid index names. For example, the following code fills a listbox with the names of all indexes defined for the *CustomersTable* dataset:
Opening and Closing Datasets (see page 1604)	To read or write data in a dataset, an application must first open it. You can open a dataset in two ways:

Posting Data (see page 1606)	After you finish editing a record, you must call the Post method to write out your changes. The *Post* method behaves differently, depending on the dataset's state and on whether you are caching updates. • If you are not caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to the database and returns the dataset to the *dsBrowse* state. • If you are caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to an internal cache and returns the dataset to the *dsBrowse* state. The edits... more (see page 1606)
Preparing Queries (see page 1606)	Preparing a query is an optional step that precedes query execution. Preparing a query submits the SQL statement and its parameters, if any, to the data access layer and the database server for parsing, resource allocation, and optimization. In some datasets, the dataset may perform additional setup operations when preparing the query. These operations improve query performance, making your application faster, especially when working with updatable queries. An application can prepare a query by setting the *Prepared* property to *True*. If you do not prepare a query before executing it, the dataset automatically prepares it for you each time... more (see page 1606)
Preparing Stored Procedures (see page 1607)	As with query-type datasets, stored procedure-type datasets must be prepared before they execute the stored procedure. Preparing a stored procedure tells the data access layer and the database server to allocate resources for the stored procedure and to bind parameters. These operations can improve performance. If you attempt to execute a stored procedure before preparing it, the dataset automatically prepares it for you, and then unprepares it after it executes. If you plan to execute a stored procedure a number of times, it is more efficient to explicitly prepare it by setting the *Prepared* property to *True*.
Searching On Partial Keys (see page 1607)	Each time you call *SetKey* or *FindKey*, the method clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey* and *FindKey*. For example, suppose you have already executed a search of the Employee table based on the City field of the "CityIndex" index. Suppose further that "CityIndex" includes both the *City* and *Company* fields. To find a record with a specified company name in a specified city, use the following code:... more (see page 1607)
Searching On Partial Keys (see page 1608)	If the dataset has more than one key column, and you want to search for values in a subset of that key, set *KeyFieldCount* to the number of columns on which you are searching. For example, if the dataset's current index has three columns, and you want to search for values using just the first column, set *KeyFieldCount* to 1. For table type datasets with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or... more (see page 1608)
Setting Filter Options (see page 1608)	The FilterOptions property lets you specify whether a filter that compares string-based fields accepts records based on partial comparisons and whether string comparisons are case-sensitive. *FilterOptions* is a set property that can be an empty set (the default), or that can contain either or both of the following values: ***FilterOptions values***
Setting the Filter Property (see page 1608)	To create a filter using the Filter property, set the value of the property to a string that contains the filter's test condition. For example, the following statement creates a filter that tests a dataset's *State* field to see if it contains a value for the state of California:

Sorting Records with Indexes (see page 1609)	An index determines the display order of records in a table. Typically, records appear in ascending order based on a primary, or default, index. This default behavior does not require application intervention. If you want a different sort order, however, you must specify either • An alternate index. • A list of columns on which to sort (not available on servers that aren't SQL-based). Indexes let you present the data from a table in different orders. On SQL-based tables, this sort order is implemented by using the index to generate an ORDER BY clause in a query that fetches the table's records.... more (see page 1609)
Specifying Ranges (see page 1610)	There are two mutually exclusive ways to specify a range: • Specify the beginning and ending separately using *SetRangeStart* and *SetRangeEnd*. • Specify both endpoints at once using *SetRange*.
Specifying the Current Record After a Successful Search (see page 1612)	By default, a successful search positions the cursor on the first record that matches the search criteria. If you prefer, you can set the *KeyExclusive* property to *True* to position the cursor on the next record after the first matching record. By default, *KeyExclusive* is *False*, meaning that successful searches position the cursor on the first matching record.
Specifying the Query (see page 1613)	For true query-type datasets, you use the *SQL* property to specify the SQL statement for the dataset to execute. Some datasets, such as *TADODataSet*, *TSQLDataSet,* and client datasets, use a *CommandText* property to accomplish the same thing. Most queries that return records are SELECT commands. Typically, they define the fields to include, the tables from which to select those fields, conditions that limit what records to include, and the order of the resulting dataset. For example:
Synchronizing Tables (see page 1614)	If you have two or more datasets that represent the same database table but do not share a data source component, then each dataset has its own view on the data and its own current record. As users access records through each datasets, the components' current records will differ. If the datasets are all instances of *TTable*, or all instances of *TIBTable*, or all client datasets, you can force the current record for each of these datasets to be the same by calling the *GotoCurrent* method. *GotoCurrent* sets its own dataset's current record to the current record of... more (see page 1614)
Understanding the Differences Between Ranges and Filters (see page 1615)	Both ranges and filters restrict visible records to a subset of all available records, but the way they do so differs. A range is a set of contiguously indexed records that fall between specified boundary values. For example, in an employee database indexed on last name, you might apply a range to display all employees whose last names are greater than "Jones" and less than "Smith". Because ranges depend on indexes, you must set the current index to one that can be used to define the range. As with specifying an index to sort records (see page 1609), you can assign the... more (see page 1615)
Using Indexes to Search for Records (see page 1615)	You can search against any dataset using the Locate and Lookup methods of *TDataSet*. However, by explicitly using indexes, some table type datasets can improve over the searching performance provided by the *Locate* and *Lookup* methods. ADO datasets all support the Seek method, which moves to a record based on a set of field values for fields in the current index. *Seek* lets you specify where to move the cursor relative to the first or last matching record. *TTable* and all types of client dataset support similar indexed-based searches, but use a combination of related methods. The following table... more (see page 1615)
Using Nested Detail Tables (see page 1616)	A nested table is a detail dataset that is the value of a single dataset field in another (master) dataset. For datasets that represent server data, a nested detail dataset can only be used for a dataset field on the server. *TClientDataSet* components do not represent server data, but they can also contain dataset fields if you create a dataset for them that contains nested details, or if they receive data from a provider that is linked to the master table of a master/detail relationship. **Note:** For *TClientDataSet*, using nested detail sets is necessary if you want to apply... more (see page 1616)

Using Parameters in Queries (see page 1617)	A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values, such as those used in a WHERE clause for comparisons, that appear in an SQL statement. Ordinarily, parameters stand in for data values passed to the statement. For example, in the following INSERT statement, values to insert are passed as parameters:
Using Query-type Datasets (see page 1618)	
Using Stored Procedure-type Datasets (see page 1618)	How your application uses a stored procedure depends on how the stored procedure was coded, whether and how it returns data, the specific database server used, or a combination of these factors.
Using TDataSet Descendants (see page 1619)	*TDataSet* has several immediate descendants, each of which corresponds to a different data access mechanism. You do not work directly with any of these descendants. Rather, each descendant introduces the properties and methods for using a particular data access mechanism. These properties and methods are then exposed by descendant classes that are adapted to different types of server data. The immediate descendants of *TDataSet* include • *TBDEDataSet*, which uses the Borland Database Engine (BDE) to communicate with the database server. The *TBDEDataSet* descendants you use are *TTable*, *TQuery*, *TStoredProc*, and *TNestedTable*. The unique features of... more (see page 1619)
Using Table Type Datasets (see page 1620)	
Using the Eof and Bof Properties (see page 1621)	Two read-only, runtime properties, *Eof* (End-of-file) and *Bof(Beginning-of-file), are useful when you want to iterate through all records in a dataset.*
Using the First and Last Methods (see page 1623)	The First method moves the cursor to the first row in a dataset and sets the BOF property to *True*. If the cursor is already at the first row in the dataset, *First* does nothing. For example, the following code moves to the first record in *CustTable*:
Using the MoveBy Method (see page 1623)	MoveBy lets you specify how many rows forward or back to move the cursor in a dataset. Movement is relative to the current record at the time that *MoveBy* is called. *MoveBy* also sets the BOF and EOF properties for the dataset as appropriate. This function takes an integer parameter, the number of records to move. Positive integers indicate a forward move and negative integers indicate a backward move. **Note:** *MoveBy* raises an exception in unidirectional datasets if you use a negative argument. *MoveBy* returns the number of rows it moves. If you attempt to move past the beginning or... more (see page 1623)
Using the Next and Prior Methods (see page 1624)	The Next method moves the cursor forward one row in the dataset and sets the BOF property to *False* if the dataset is not empty. If the cursor is already at the last row in the dataset when you call *Next*, nothing happens. For example, the following code moves to the next record in *CustTable*:
Using Unidirectional Result Sets (see page 1624)	When a query-type dataset returns a result set, it also receives a cursor, or pointer to the first record in that result set. The record pointed to by the cursor is the currently active record. The current record is the one whose field values are displayed in data-aware components associated with the result set's data source. Unless you are using dbExpress, this cursor is bi-directional by default. A bi-directional cursor can navigate both forward and backward through its records. Bi-directional cursor support requires some additional processing overhead, and can slow some queries. If you do not need to be able... more (see page 1624)

Working with Stored Procedure Parameters (see page 1625)	There are four types of parameters that can be associated with stored procedures: • *Input p*arameters, used to pass values to a stored procedure for processing. • *Output* parameters, used by a stored procedure to pass return values to an application. • *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application. • *A result parameter*, used by some stored procedures to return an error or status value to an application. A stored procedure can only return one result parameter. Whether a stored procedure uses a... more (see page 1625)
Using Locate (see page 1627)	Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. (Partial-key matching is when the criterion string need only be a prefix of the field value.) For example, the following code moves the cursor to the first row in the *CustTable* where the value in the *Company* column is "Professional Divers, Ltd.":
Using Lookup (see page 1628)	Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it. In its simplest form, you pass *Lookup* the name of field to search, the field value to match, and the field or fields to return. For example, the following code looks for the first record in the *CustTable* where the value of... more (see page 1628)
Modifying Data (see page 1629)	You can use the following dataset methods to insert, update, and delete data if the read-only *CanModify* property is *True*. *CanModify* is *True* unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor intervenes. (Intervening factors include the *ReadOnly* property on some datasets or the *RequestLive* property on *TQuery* components.) ***Dataset methods for inserting, updating, and deleting data***
Searching Datasets (see page 1630)	If a dataset is not unidirectional, you can search against it using the *Locate* and *Lookup* methods. These methods enable you to search on any type of columns in any dataset. The following topics discuss Locate and Lookup in greater detail: • Using Locate (see page 1627) • Using Lookup (see page 1628) **Note:** Some *TDataSet* descendants introduce an additional family of methods for searching based on an index. For information about these additional methods, see Using Indexes to Search for Records (see page 1615).
Specifying an Index with IndexName (see page 1630)	Use the *IndexName* property to cause an index to be active. Once active, an index determines the order of records in the dataset. (It can also be used as the basis for a master-detail link, an index-based search, or index-based filtering.) To activate an index, set the *IndexName* property to the name of the index. In some database systems, primary indexes do not have names. To activate one of these indexes, set *IndexName* to a blank string. At design-time, you can select an index from a list of available indexes by clicking the property's ellipsis button in the **Object Inspector... more (see page 1630)**

Displaying and Editing a Subset of Data Using Filters (see page 1631)	An application is frequently interested in only a subset of records from a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values. In each case, you can use filters to restrict an application's access to a subset of all records in the dataset. With unidirectional datasets, you can only limit the records in the dataset by using a query that restricts the records in the dataset. With other *TDataSet* descendants,... more (see page 1631)
Understanding Datasets: Overview (see page 1632)	The fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. A dataset object represents a set of records from a database organized into a logical table. These records may be the records from a single database table, or they may represent the results of executing a query or stored procedure. All dataset objects that you use in your database applications descend from TDataSet, and they inherit data fields, properties, events, and methods from this class. *TDataSet* is a virtualized dataset, meaning that many of its properties and methods are **virtual... more (see page 1632)**
Determining Dataset States (see page 1633)	The *state*—or *mode*—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. At runtime, you can examine a dataset's read-only State property to determine its current state. The following table summarizes possible values for the *State* property and what they mean: ***Values for the dataset State property***
Navigating Datasets (see page 1634)	Each active dataset has a *cursor*, or pointer, to the current row in the dataset. The *current row* in a dataset is the one whose field values currently show in single-field, data-aware controls on a form, such as TDBEdit, *TDBLabel*, and TDBMemo. If the dataset supports editing, the current record contains the values that can be manipulated by edit, insert, and delete methods. You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records: ***Navigational methods of... more (see page 1634)***
Writing an OnFilterRecord Event Handler (see page 1635)	You can write code to filter records using the OnFilterRecord events generated by the dataset for each record it retrieves. This event handler implements a test that determines if a record should be included in those that are visible to the application. To indicate whether a record passes the filter condition, your *OnFilterRecord* handler sets its *Accept* parameter to *True* to include a record, or *False* to exclude it. For example, the following filter displays only those records with the State field set to "CA":
Types of Datasets (see page 1635)	Using TDataSet descendants (see page 1619) classifies *TDataSet* descendants by the method they use to access their data. Another useful way to classify *TDataSet* descendants is to consider the type of server data they represent. Viewed this way, there are three basic classes of datasets: Table type datasets (see page 1620): Table type datasets represent a single table from the database server, including all of its rows and columns. Table type datasets include TTable, TADOTable, TSQLTable, and TIBTable. Table type datasets let you take advantage of indexes defined on the server. Because there is a one-to-one correspondence between database table and dataset, you can use... more (see page 1635)

3.2.3.6.1 Creating an Index with IndexFieldNames

If there is no defined index that implements the sort order you want, you can create a pseudo-index using the *IndexFieldNames* property.

Note: *IndexName and IndexFieldNames are mutually exclusive. Setting one property clears values set for the other. For information about IndexName, see Specifying an index with IndexName (see page 1630).*

The value of IndexFieldNames is a string. To specify a sort order, list each column name to use in the order it should be used, and delimit the names with semicolons. Sorting is by ascending order only. Case-sensitivity of the sort depends on the capabilities of your server. See your server documentation for more information.

The following code sets the sort order for PhoneTable based on LastName, then FirstName:

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

```
PhoneTable->IndexFieldNames = "LastName;FirstName";
```

Note: If you use *IndexFieldNames* on Paradox and dBASE tables, the dataset attempts to find an index that uses the columns you specify. If it cannot find such an index, it raises an exception.

See Also

Obtaining Information About Indexes (see page 1603)

Specifying an Index with IndexName (see page 1630)

IndexFieldNames

IndexFieldNames

IndexFieldNames

IndexFieldNames

IndexFieldNames

IndexFieldNames

3.2.3.6.2 Supplying Parameters at Design Time

At design time, you can specify parameter values using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the **Object Inspector**. If the SQL statement does not contain any parameters, no objects are listed in the collection editor.

Note: The parameter collection editor is the same collection editor that appears for other collection properties. Because the editor is shared with other properties, its right-click context menu contains the Add and Delete commands. However, they are never enabled for query parameters. The only way to add or delete parameters is in the SQL statement itself.

For each parameter, select it in the parameter collection editor. Then use the **Object Inspector** to modify its properties.

When using the *Params* property (*TParam* objects), you will want to inspect or modify the following,

The DataType property lists the data type for the parameter's value. For some datasets, this value may be correctly initialized. If the dataset did not deduce the type, *DataType* is *ftUnknown*, and you must change it to indicate the type of the parameter value.

The *DataType* property lists the logical data type for the parameter. In general, these data types conform to server data types. For specific logical type-to-server data type mappings, see the documentation for the data access mechanism (BDE, dbExpress, InterBase).

The ParamType property lists the type of the selected parameter. For queries, this is always *ptInput*, because queries can only contain input parameters. If the value of *ParamType* is *ptUnknown*, change it to *ptInput*.

The Value property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

When using the *Parameters* property (*TParameter* objects), you will want to inspect or modify the following:

The DataType property lists the data type for the parameter's value. For some data types, you must provide additional information:

- The NumericScale property indicates the number of decimal places for numeric parameters.
- The Precision property indicates the total number of digits for numeric parameters.
- The Size property indicates the number of characters in string parameters.

The Direction property lists the type of the selected parameter. For queries, this is always *pdInput*, because queries can only contain input parameters.

The Attributes property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.

The Value property specifies a value for the selected parameter. You can leave *Value* blank if your application supplies parameter values at runtime.

See Also

Supplying Parameters at Runtime (see page 1583)

Establishing Master/detail Relationships Using Parameters (see page 1593)

3.2.3.6.3 Supplying Parameters at Runtime

To create parameters at runtime, you can use the

- *ParamByName* method to assign values to a parameter based on its name (not available for *TADOQuery*)
- *Params* or *Parameters* property to assign values to a parameter based on the parameter's ordinal position within the SQL statement.
- *Params.ParamValues* or *Parameters.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set.

The following code uses *ParamByName* to assign the text of an edit box to the :Capital parameter:

```
SQLQuery1.ParamByName('Capital').AsString := Edit1.Text;
SQLQuery1->ParamByName("Capital")->AsString = Edit1->Text;
```

The same code can be rewritten using the *Params* property, using an index of 0 (assuming the :Capital parameter is the first parameter in the SQL statement):

```
SQLQuery1.Params[0].AsString := Edit1.Text;
SQLQuery1->Params->Items[0]->AsString = Edit1->Text;
```

The command line below sets three parameters at once, using the *Params.ParamValues* property:

```
Query1.Params.ParamValues['Name;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
Query1->Params->ParamValues["Name;Capital;Continent"] =
  VarArrayOf(OPENARRAY(Variant, (Edit1->Text, Edit2->Text, Edit3->Text)));
```

Note that *ParamValues* uses Variants, avoiding the need to cast values.

See Also

Supplying Parameters at Design Time (see page 1582)

Establishing Master/detail Relationships Using Parameters (see page 1593)

3.2.3.6.4 Adding New Records

A dataset must be in *dsInsert* mode before an application can add new records. In code, you can use the Insert or Append methods to put a dataset into *dsInsert* mode if the read-only CanModify property for the dataset is *True*.

When a dataset transitions to *dsInsert* mode, it first receives a *BeforeInsert* event. After the transition to insert mode is successfully completed, the dataset receives first an *OnNewRecord* event hand then an *AfterInsert* event. You can use these events, for example, to provide initial values to newly inserted records:

```
procedure TForm1.OrdersTableNewRecord(DataSet: TDataSet);
begin
  DataSet.FieldByName('OrderDate').AsDateTime := Date;
end;
void __fastcall TForm1::OrdersTableNewRecord(TDataSet *DataSet)
{
```

```
  DataSet->FieldByName("OrderDate")->AsDateTime = Date();
}
```

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *False* (the default), and
- *CanModify* is *True* for the dataset.

Note: Even if a dataset is in *dsInsert* state, adding records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsInsert* mode, a user or application can enter values into the fields associated with the new record. Except for the grid and navigational controls, there is no visible difference to a user between *Insert* and *Append*. On a call to *Insert*, an empty row appears in a grid above what was the current record. On a call to *Append*, the grid is scrolled to the last record in the dataset, an empty row appears at the bottom of the grid, and the Next and Last buttons are dimmed on any navigator component associated with the dataset.

Data-aware controls for which inserting is enabled automatically call Post when a user executes any action that changes which record is current (such as moving to a different record in a grid). Otherwise you must call *Post* in your code.

Post writes the new record to the database, or, if you are caching updates, *Post* writes the record to an in-memory cache. To write cached inserts and appends to the database, call the dataset's *ApplyUpdates* method.

Inserting records

Insert opens a new, empty record before the current record, and makes the empty record the current record so that field values for the record can be entered either by a user or by your application code.

When an application calls Post (or *ApplyUpdates* when using cached updates), a newly inserted record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is inserted into the dataset at its current position.
- For SQL databases, the physical location of the insertion is implementation-specific. If the table is indexed, the index is updated with the new record information.

Appending records

Append opens a new, empty record at the end of the dataset, and makes the empty record the current one so that field values for the record can be entered either by a user or by your application code.

When an application calls Post (or *ApplyUpdates* when using cached updates), a newly appended record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed Paradox and dBASE tables, the record is added to the end of the dataset.
- For SQL databases, the physical location of the append is implementation-specific. If the table is indexed, the index is updated with the new record information.

See Also

Editing Records (see page 1591)

Deleting Records (see page 1591)

Posting Data to the Database (see page 1606)

Canceling Changes (see page 1586)

Modifying Entire Records (see page 1601)

3.2.3.6.5 Applying or Canceling a Range

When you call *SetRangeStart* or *EditRangeStart* to specify the start of a range, or *SetRangeEnd* or *EditRangeEnd* to specify the end of a range, the dataset enters the *dsSetKey* state. It stays in that state until you apply or cancel the range.

Applying a range

When you specify a range, the boundary conditions you define are not put into effect until you apply the range. To make a range take effect, call the *ApplyRange* method. *ApplyRange* immediately restricts a user's view of and access to data in the specified subset of the dataset.

Canceling a range

The *CancelRange* method ends application of a range and restores access to the full dataset. Even though canceling a range restores access to all records in the dataset, the boundary conditions for that range are still available so that you can reapply the range at a later time. Range boundaries are preserved until you provide new range boundaries or modify the existing boundaries. For example, the following code is valid:

```
.
.
.
MyTable.CancelRange;
.
.
.
{later on, use the same range again. No need to call SetRangeStart, etc.}
MyTable.ApplyRange;
.
.
.
.
.
.
MyTable->CancelRange();
.
.
.
// later on, use the same range again. No need to call SetRangeStart, etc.
MyTable->ApplyRange();
.
.
.
```

See Also

Understanding the Differences Between Ranges and Filters (see page 1615)

Modifying a Range (see page 1601)

Setting Ranges (see page 1610)

ApplyRange

ApplyRange

CancelRange

CancelRange

3.2.3.6.6 Calculating Fields

Using the Fields editor, you can define calculated fields (see page 1872) for your datasets. When a dataset contains

calculated fields, you provide the code to calculate those field's values in an OnCalcFields event handler.

The AutoCalcFields property determines when *OnCalcFields* is called. If *AutoCalcFields* is *True*, *OnCalcFields* is called when

- A dataset is opened.
- The dataset enters edit mode.
- A record is retrieved from the database.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified.

If *AutoCalcFields* is *False*, then *OnCalcFields* is not called when individual fields within a record are edited (the fourth condition above).

Warning: *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or a linked dataset if it is part of a master-detail relationship), because this leads to recursion. For example, if *OnCalcFields* performs a Post, and *AutoCalcFields* is *True*, then *OnCalcFields* is called again, causing another Post, and so on.

When *OnCalcFields* executes, a dataset enters *dsCalcFields* mode. This state prevents modifications or additions to the records except for the calculated fields the handler is designed to modify. The reason for preventing other modifications is because *OnCalcFields* uses the values in other fields to derive calculated field values. Changes to those other fields might otherwise invalidate the values assigned to calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

See Also

Types of Datasets (see page 1635)

Opening and Closing Datasets (see page 1604)

Determining Dataset States (see page 1633)

Navigating Datasets (see page 1634)

Searching Datasets (see page 1630)

Displaying and Editing a Subset of Data Using Filters (see page 1631)

Modifying Data (see page 1629)

Using TDataSet Descendants (see page 1619)

3.2.3.6.7 Canceling Changes

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called Post. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the Cancel method for the dataset. A call to Cancel always returns a dataset to *dsBrowse* state.

If the dataset was in *dsEdit* or *dsInsert* mode when your application called *Cancel*, it receives *BeforeCancel* and *AfterCancel* events before and after the current record is restored to its original values.

On forms, you can allow users to cancel edit, insert, or append operations by including the Cancel button on a navigator component associated with the dataset, or you can provide code for your own Cancel button on the form.

See Also

Editing Records (see page 1591)

Adding New Records (see page 1583)

Deleting Records (see page 1591)

Posting Data to the Database (see page 1606)

Modifying Entire Records (see page 1601)

3.2.3.6.8 Controlling Read/Write Access to Tables

By default when a table type dataset is opened, it requests read and write access for the underlying database table. Depending on the characteristics of the underlying database table, the requested write privilege may not be granted (for example, when you request write access to an SQL table on a remote server and the server restricts the table's access to read only).

Note: This is not true for *TClientDataSet*, which determines whether users can edit data from information that the dataset provider supplies with data packets. It is also not true for *TSQLTable*, which is a unidirectional dataset, and hence always read-only.

When the table opens, you can check the CanModify property to ascertain whether the underlying database (or the dataset provider) allows users to edit the data in the table. If *CanModify* is *False*, the application cannot write to the database. If *CanModify* is *True*, your application can write to the database provided the table's *ReadOnly* property is *False*.

ReadOnly determines whether a user can both view and edit data. When *ReadOnly* is *False* (the default), a user can both view and edit data. To restrict a user to viewing data, set *ReadOnly* to *True* before opening the table.

Note: *ReadOnly* is implemented on all table type datasets except *TSQLTable*, which is always read-only.

See Also

Sorting Records with Indexes (see page 1609)

Limiting Records with Ranges (see page 1597)

Using Indexes to Search for Records (see page 1615)

Creating Master/detail Relationships (see page 1590)

Creating and Deleting Tables (see page 1587)

Emptying Tables (see page 1592)

Synchronizing Tables (see page 1614)

3.2.3.6.9 Creating and Deleting Tables

Some table type datasets let you create and delete the underlying tables at design time or at runtime. Typically, database tables are created and deleted by a database administrator. However, it can be handy during application development and testing to create and destroy database tables that your application can use.

Creating tables

TTable and *TIBTable* both let you create the underlying database table without using SQL. Similarly, *TClientDataSet* lets you create a dataset when you are not working with a dataset provider. Using *TTable* and *TClientDataSet*, you can create the table at design time or runtime. *TIBTable* only lets you create tables at runtime.

Before you can create the table, you must be set properties to specify the structure of the table you are creating. In particular, you must specify

- The database that will host the new table. For *TTable*, you specify the database using the DatabaseName property. For *TIBTable*, you must use a *TIBDatabase* component, which is assigned to the Database property. (Client datasets do not use a database.)
- The type of database (*TTable* only). Set the TableType property to the desired type of table. For Paradox, dBASE, or ASCII tables, set *TableType* to *ttParadox*, *ttDBase*, or *ttASCII*, respectively. For all other table types, set *TableType* to *ttDefault*.

- The name of the table you want to create. Both *TTable* and *TIBTable* have a *TableName* property for the name of the new table. Client datasets do not use a table name, but you should specify the *FileName* property before you save the new table. If you create a table that duplicates the name of an existing table, the existing table and all its data are overwritten by the newly created table. The old table and its data cannot be recovered. To avoid overwriting an existing table, you can check the *Exists* property at runtime. *Exists* is only available on *TTable* and *TIBTable*.
- Indexes for the new table (optional). At design time, double-click the *IndexDefs* property in the **Object Inspector** to bring up the collection editor. Use the collection editor to add, remove, or change the properties of index definitions. At runtime, clear any existing index definitions, and then use the AddIndexDef method to add each new index definition. For each new index definition, set the properties of the TIndexDef object to specify the desired attributes of the index.
- The fields for the new table. There are two ways to do this:
- You can add field definitions to the FieldDefs property. At design time, double-click the *FieldDefs* property in the **Object Inspector** to bring up the collection editor. Use the collection editor to add, remove, or change the properties of the field definitions. At runtime, clear any existing field definitions and then use the AddFieldDef method to add each new field definition. For each new field definition, set the properties of the TFieldDef object to specify the desired attributes of the field.
- You can use persistent field components instead. At design time, double-click on the dataset to bring up the *Fields* editor. In the Fields editor, right-click and choose the New Field command. Describe the basic properties of your field. Once the field is created, you can alter its properties in the **Object Inspector** by selecting the field in the Fields editor.

  Note: You can't define indexes for the new table if you are using persistent field components instead of field definition objects.

  To create the table at design time, right-click the dataset and choose Create Table (*TTable*) or Create Data Set (*TClientDataSet*). This command does not appear on the context menu until you have specified all the necessary information.

  To create the table at runtime, call the *CreateTable* method (*TTable* and *TIBTable*) or the *CreateDataSet* method (*TClientDataSet*).

  Note: You can set up the definitions at design time and then call the *CreateTable* (or *CreateDataSet*) method at runtime to create the table. However, to do so you must indicate that the definitions specified at runtime should be saved with the dataset component. (by default, field and index definitions are generated dynamically at runtime). Specify that the definitions should be saved with the dataset by setting its *StoreDefs* property to *True*.

  Tip: If you are using *TTable*, you can preload the field definitions and index definitions of an existing table at design time. Set the *DatabaseName* and *TableName* properties to specify the existing table. Right click the table component and choose Update Table Definition. This automatically sets the values of the *FieldDefs* and *IndexDefs* properties to describe the fields and indexes of the existing table. Next, reset the *DatabaseName* and *TableName* to specify the table you want to create, canceling any prompts to rename the existing table.

  Note: When creating Oracle8 tables, you can't create object fields (ADT fields, array fields, and dataset fields).

  The following code creates a new table at runtime and associates it with the DBDEMOS alias. Before it creates the new table, it verifies that the table name provided does not match the name of an existing table:

```
var
  TableFound: Boolean;
begin
  with TTable.Create(nil) do // create a temporary TTable component
  begin
    try
      { set properties of the temporary TTable component }
      Active := False;
      DatabaseName := 'DBDEMOS';
      TableName := Edit1.Text;
      TableType := ttDefault;
      { define fields for the new table }
      FieldDefs.Clear;
      with FieldDefs.AddFieldDef do begin
        Name := 'First';
        DataType := ftString;
        Size := 20;
```

```
        Required := False;
      end;
      with FieldDefs.AddFieldDef do begin
        Name := 'Second';
        DataType := ftString;
        Size := 30;
        Required := False;
      end;
      { define indexes for the new table }
      IndexDefs.Clear;
      with IndexDefs.AddIndexDef do begin
        Name := '';
        Fields := 'First';
        Options := [ixPrimary];
      end;
      TableFound := Exists; // check whether the table already exists
      if TableFound then
        if MessageDlg('Overwrite existing table ' + Edit1.Text + '?',
             mtConfirmation, mbYesNoCancel, 0) = mrYes then
          TableFound := False;
      if not TableFound then
        CreateTable; // create the table
    finally
      Free; // destroy the temporary TTable when done
    end;
  end;
end;
TTable *NewTable = new TTable(Form1);
NewTable->Active = false;
NewTable->DatabaseName = "BCDEMOS";
NewTable->TableName = Edit1->Text;
NewTable->TableType = ttDefault;
NewTable->FieldDefs->Clear();
TFieldDef *NewField = NewTable->FieldDefs->AddFieldDef(); // define first field
NewField->DataType = ftInteger;
NewField->Name = Edit2->Text;
NewField = NewTable->FieldDefs->AddFieldDef(); // define second field
NewField->DataType = ftString;
NewField->Size = StrToInt(Edit3->Text);
NewField->Name = Edit4->Text;
NewTable->IndexDefs->Clear();
TIndexDef *NewIndex = NewTable->IndexDefs->AddIndexDef(); // add an index
NewIndex->Name = "PrimaryIndex";
NewIndex->Fields = Edit2->Text;
NewIndex->Options << ixPrimary << ixUnique;
// Now check for prior existence of this table
bool CreateIt = (!NewTable->Exists);
if (!CreateIt)
   if (Application->MessageBox((AnsiString("Overwrite table ") + Edit1->Text +
                                AnsiString("?")).c_str(),
                                "Table Exists", MB_YESNO) == IDYES)
    CreateIt = true;
if (CreateIt)
  NewTable->CreateTable(); // create the table
```

Deleting tables

TTable and *TIBTable* let you delete tables from the underlying database table without using SQL. To delete a table at runtime, call the dataset's *DeleteTable* method. For example, the following statement removes the table underlying a dataset:

```
CustomersTable.DeleteTable;
CustomersTable->DeleteTable();
```

Warning: When you delete a table with *DeleteTable*, the table and all its data are gone forever.

If you are using *TTable*, you can also delete tables at design time: Right-click the table component and select Delete Table from the context menu. The Delete Table menu pick is only present if the table component represents an existing database table (the

DatabaseName and TableName properties specify an existing table).

See Also

Sorting Records with Indexes (see page 1609)

Limiting Records with Ranges (see page 1597)

Using Indexes to Search for Records (see page 1615)

Creating Master/detail Relationships (see page 1590)

Controlling Read/write Access to Tables (see page 1587)

Emptying Tables (see page 1592)

Synchronizing Tables (see page 1614)

3.2.3.6.10 Creating Filters

There are two ways to create a filter for a dataset:

- Set the Filter property (see page 1608). Filter is especially useful for creating and applying filters at runtime.
- Write an OnFilterRecord event handler (see page 1635) for simple or complex filter conditions. With OnFilterRecord, you specify filter conditions at design time. Unlike the *Filter* property, which is restricted to a single string containing filter logic, an *OnFilterRecord* event can take advantage of branching and looping logic to create complex, multi-level filter conditions.

The main advantage to creating filters using the *Filter* property is that your application can create, change, and apply filters dynamically, (for example, in response to user input). Its main disadvantages are that filter conditions must be expressible in a single text string, cannot make use of branching and looping constructs, and cannot test or compare its values against values not already in the dataset.

The strengths of the *OnFilterRecord* event are that a filter can be complex and variable, can be based on multiple lines of code that use branching and looping constructs, and can test dataset values against values outside the dataset, such as the text in an edit box. The main weakness of using *OnFilterRecord* is that you set the filter at design time and it cannot be modified in response to user input. (You can, however, create several filter handlers and switch among them in response to general application conditions.)

The following sections describe how to create filters using the *Filter* property and the *OnFilterRecord* event handler.

See Also

Setting Filter Options (see page 1608)

3.2.3.6.11 Creating Master/detail Relationships

Table type datasets can be linked into master/detail relationships. When you set up a master/detail relationship, you link two datasets so that all the records of one (the detail) always correspond to the single current record in the other (the master).

Table type datasets support master/detail relationships in two very distinct ways:

- All table type datasets can act as the detail of another dataset by linking cursors. This process is described in Making the table a detail of another dataset (see page 1598).
- *TTable*, *TSQLTable*, and all client datasets can act as the master in a master/detail relationship that uses nested detail tables. This process is described in Using nested detail tables (see page 1616).

Each of these approaches has its unique advantages. Linking cursors lets you create master/detail relationships where the master table is any type of dataset. With nested details, the type of dataset that can act as the detail table is limited, but they provide for more options in how to display the data. If the master is a client dataset, nested details provide a more robust mechanism for applying cached updates.

See Also

Sorting Records with Indexes (see page 1609)

Limiting Records with Ranges (see page 1597)

Using Indexes to Search for Records (see page 1615)

Controlling Read/write Access to Tables (see page 1587)

Creating and Deleting Tables (see page 1587)

Emptying Tables (see page 1592)

Synchronizing Tables (see page 1614)

3.2.3.6.12 Deleting Records

Use the Delete method to delete the current record in an active dataset. When the *Delete* method is called,

- The dataset receives a *BeforeDelete* event.
- The dataset attempts to delete the current record.
- The dataset returns to the *dsBrowse* state.
- The dataset receives an *AfterDelete* event.

If want to prevent the deletion in the *BeforeDelete* event handler, you can call the global *Abort* procedure:

```
procedure TForm1.TableBeforeDelete (Dataset: TDataset)
begin
  if MessageDlg('Delete This Record?', mtConfirmation, mbYesNoCancel, 0) <> mrYes then
    Abort;
end;
void __fastcall TForm1::TableBeforeDelete (TDataSet *Dataset)
{
  if (MessageBox(0, "Delete This Record?", "CONFIRM", MB_YESNO) != IDYES)
Abort();
}
```

If *Delete* fails, it generates an *OnDeleteError* event. If the *OnDeleteError* event handler can't correct the problem, the dataset remains in *dsEdit* state. If *Delete* succeeds, the dataset reverts to the *dsBrowse* state and the record that followed the deleted record becomes the current record.

If you are caching updates, the deleted record is not removed from the underlying database table until you call *ApplyUpdates*.

If you provide a navigator component on your forms, users can delete the current record by clicking the navigator's Delete button. In code, you must call *Delete* explicitly to remove the current record.

See Also

Editing Records (see page 1591)

Adding New Records (see page 1583)

Posting Data to the Database (see page 1606)

Canceling Changes (see page 1586)

Modifying Entire Records (see page 1601)

3.2.3.6.13 Editing Records

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit method to put a*

dataset into dsEdit mode if the read-only CanModify property for the dataset is True.

When a dataset transitions to dsEdit mode, it first receives a BeforeEdit event. After the transition to edit mode is successfully completed, the dataset receives an AfterEdit event. Typically, these events are used for updating the user interface to indicate the current state of the dataset. If the dataset can't be put into edit mode for some reason, an OnEditError event occurs, where you can inform the user of the problem or try to correct the situation that prevented the dataset from entering edit mode.

On forms in your application, some data-aware controls can automatically put a dataset into dsEdit state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Note: Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsEdit* mode, a user can modify the field values for the current record that appears in any data-aware controls on a form. Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

If you have a navigator component on your form, users can cancel edits by clicking the navigator's Cancel button. Canceling edits returns a dataset to *dsBrowse* state.

In code, you must write or cancel edits by calling the appropriate methods. You write changes by calling Post. You cancel them by calling Cancel. In code, Edit and Post are often used together. For example,

```
with CustTable do
begin
  Edit;
  FieldValues['CustNo'] := 1234;
  Post;
end;
```

```
Table1->Edit();
Table1->FieldValues["CustNo"] = 1234;
Table1->Post();
```

In the previous example, the first line of code places the dataset in *dsEdit* mode. The next line of code assigns the number 1234 to the *CustNo* field of the current record. Finally, the last line writes, or posts, the modified record. If you are not caching updates, posting writes the change back to the database. If you are caching updates, the change is written to a temporary buffer, where it stays until the dataset's *ApplyUpdates* method is called.

See Also

Adding New Records (see page 1583)

Deleting Records (see page 1591)

Posting Data to the Database (see page 1606)

Canceling Changes (see page 1586)

Modifying Entire Records (see page 1601)

3.2.3.6.14 Emptying Tables

Many table type datasets supply a single method that lets you delete all rows of data in the table.

Table Type	Method
TTable and *TIBTable*	You can delete all the records by calling the *EmptyTable* method at runtime: PhoneTable.EmptyTable; PhoneTable->EmptyTable();
TADOTable	You can use the *DeleteRecords* method: PhoneTable.DeleteRecords; PhoneTable->DeleteRecords(arAll);

TSQLTable	You can use the *DeleteRecords* method. Note, that the *TSQLTable* version of *DeleteRecords* never takes any parameters: PhoneTable.DeleteRecords; PhoneTable->DeleteRecords();
EmptyDataSet	For client datasets, you can use the *EmptyDataSet* method: PhoneTable.EmptyDataSet; PhoneTable->EmptyDataSet();

Note: For tables on SQL servers, these methods only succeed if you have DELETE privilege for the table.

Warning: When you empty a dataset, the data you delete is gone forever.

See Also

Sorting Records with Indexes (see page 1609)

Limiting Records with Ranges (see page 1597)

Using Indexes to Search for Records (see page 1615)

Creating Master/detail Relationships (see page 1590)

Controlling Read/write Access to Tables (see page 1587)

Creating and Deleting Tables (see page 1587)

Synchronizing Tables (see page 1614)

3.2.3.6.15 Enabling and Disabling Filtering

To enable filters on a dataset

1. Create a filter (see page 1590).
2. Set filter options (see page 1608) for string-based filter tests, if necessary.
3. Set the Filtered property to *True*.

When filtering is enabled, only those records that meet the filter criteria are available to an application. Filtering is always a temporary condition. You can turn off filtering by setting the *Filtered* property to *False*.

See Also

Navigating Records in a Filtered Dataset (see page 1603)

3.2.3.6.16 Establishing Master/detail Relationships Using Parameters

To set up a master/detail relationship where the detail set is a query-type dataset, you must specify a query that uses parameters. These parameters refer to current field values on the master dataset. Because the current field values on the master dataset change dynamically at runtime, you must rebind the detail set's parameters every time the master record changes. Although you could write code to do this using an event handler, all query-type datasets except *TIBQuery* provide an easier mechanism using the DataSource property.

If parameter values for a parameterized query are not bound at design time or specified at runtime, query-type datasets attempt to supply values for them based on the *DataSource* property. *DataSource* identifies a different dataset that is searched for field names that match the names of unbound parameters. This search dataset can be any type of dataset. The search dataset must be created and populated before you create the detail dataset that uses it. If matches are found in the search dataset, the detail dataset binds the parameter values to the values of the fields in the current record pointed to by the data source.

To illustrate how this works, consider two tables: a customer table and an orders table. For every customer, the orders table contains a set of orders that the customer made. The Customer table includes an ID field that specifies a unique customer ID. The orders table includes a CustID field that specifies the ID of the customer who made an order.

To set up the Customer dataset

1. Add a table type dataset to your application and bind it to the Customer table.
2. Add a *TDataSource* component named *CustomerSource*. Set its *DataSet* property to the dataset added in step 1. This data source now represents the Customer dataset.
3. Add a query-type dataset and set its *SQL* property to

```
SELECT CustID, OrderNo, SaleDate
FROM Orders
WHERE CustID = :ID
```

Note that the name of the parameter is the same as the name of the field in the master (Customer) table.

4. Set the detail dataset's *DataSource* property to *CustomerSource*. Setting this property makes the detail dataset a linked query.

At runtime the *:ID* parameter in the SQL statement for the detail dataset*is not assigned a value, so the dataset tries to match the parameter by name against a column in the dataset identified by CustomersSource. CustomersSource gets its data from the master dataset, which, in turn, derives its data from the Customer table. Because the Customer table contains a column called "ID," the value from the ID field in the current record of the master dataset is assigned to the :ID parameter for the detail dataset's SQL statement. The datasets are linked in a master-detail relationship. Each time the current record changes in the Customers dataset, the detail dataset's SELECT statement executes to retrieve all orders based on the current customer id.*

See Also

Specifying the Query (see page 1613)

Using Parameters in Queries (see page 1617)

Preparing Queries (see page 1606)

Executing Queries That Don't Return a Result Set (see page 1596)

Using Unidirectional Result Sets (see page 1624)

3.2.3.6.17 Executing a Search with Find Methods

The *Find* methods do the same thing as the Goto methods (see page 1595), except that you do not need to explicitly put the dataset in *dsSetKey* state to specify the key field values on which to search.

To execute a search using Find methods

1. Specify the index to use for the search. This is the same index that sorts the records in the dataset (see page 1609). To specify the index, use the *IndexName* or *IndexFieldNames* property.
2. Open the dataset.
3. Search for and move to the first or nearest record with *FindKey* or *FindNearest*. Both methods take a single parameter, a comma-delimited list of field values, where each value corresponds to an indexed column in the underlying table.

   Note: *FindNearest* can only be used for string fields.

See Also

Executing a Search with Goto Methods (see page 1595)

Specifying the Current Record After a Successful Search (see page 1612)

Searching on Partial Keys (see page 1608)

Searching on Partial Keys (see page 1607)

FindKey

FindKey

FindNearest

FindNearest

3.2.3.6.18 Executing a Search with Goto Methods

To execute a search using Goto methods

1. Specify the index to use for the search. This is the same index that sorts the records in the dataset (see page 1609). To specify the index, use the *IndexName* or *IndexFieldNames* property.
2. Open the dataset.
3. Put the dataset in *dsSetKey* state by calling the *SetKey* method.
4. Specify the value(s) to search on in the Fields property. *Fields* is a *TFields* object, which maintains an indexed list of field components you can access by specifying ordinal numbers corresponding to columns. The first column number in a dataset is 0.
5. Search for and move to the first matching record found with *GotoKey* or *GotoNearest*.

For example, the following code, attached to a button's *OnClick* event, uses the *GotoKey* method to move to the first record where the first field in the index has a value that exactly matches the text in an edit box:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  ClientDataSet1.SetKey;
  ClientDataSet1.Fields[0].AsString := Edit1.Text;
  if not ClientDataSet1.GotoKey then
    ShowMessage('Record not found');
end;
void __fastcall TSearchDemo::SearchExactClick(TObject *Sender)
{
  ClientDataSet1->SetKey();
  ClientDataSet1->Fields->Fields[0]->AsString = Edit1->Text;
  if (!ClientDataSet1->GotoKey())
    ShowMessage("Record not found");
}
```

GotoNearest is similar. It searches for the nearest match to a partial field value. It can be used only for string fields. For example,

```
Table1.SetKey;
Table1.Fields[0].AsString := 'Sm';
Table1.GotoNearest;
Table1->SetKey();
Table1->Fields->Fields[0]->AsString = "Sm";
Table1->GotoNearest();
```

If a record exists with "Sm" as the first two characters of the first indexed field's value, the cursor is positioned on that record. Otherwise, the position of the cursor does not change and *GotoNearest* returns *False*.

See Also

Executing a Search with Find Methods (see page 1594)

Specifying the Current Record After a Successful Search (see page 1612)

Searching on Partial Keys (see page 1608)

Reapeating or Extending a Search (see page 1607)

SetKey

SetKey

GotoKey

GotoKey

GotoNearest

GotoNearest

3.2.3.6.19 Executing Queries That Don't Return a Result Set

When a query returns a set of records (such as a SELECT query), you execute the query the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often SQL commands do not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records).

For all query-type datasets, you can execute a query that does not return a result set by calling *ExecSQL*:

```
CustomerQuery.ExecSQL;  { query does not return a result set }
CustomerQuery->ExecSQL(); // Does not return a result set
```

Tip: If you are executing the query multiple times, it is a good idea to set the *Preparedproperty to True.*

Although the query does not return any records, you may want to know the number of records it affected (for example, the number of records deleted by a DELETE query). The RowsAffected property gives the number of affected records after a call to ExecSQL.

Tip: *When you do not know at design time whether the query returns a result set (for example, if the user supplies the query dynamically at runtime), you can code both types of query execution statements in a* ***try...except*** *block. Put a call to the Open method in the* ***try*** *clause. An action query is executed when the query is activated with the Open method, but an exception occurs in addition to that. Check the exception, and suppress it if it merely indicates the lack of a result set. (For example, TQuery indicates this by an ENoResultSet exception.)*

See Also

Specifying the Query (see page 1613)

Using Parameters in Queries (see page 1617)

Establishing Master/detail Relationships Using Parameters (see page 1593)

Preparing Queries (see page 1606)

Using Unidirectional Result Sets (see page 1624)

3.2.3.6.20 Executing Stored Procedures That Don't Return a Result Set

When a stored procedure returns a cursor, you execute it the same way you populate any dataset with records: by setting *Active* to *True* or calling the *Open* method.

However, often stored procedures do not return any data, or only return results in output parameters. You can execute a stored procedure that does not return a result set by calling *ExecProc*. After executing the stored procedure, you can use the *ParamByName* method to read the value of the result parameter or of any output parameters:

```
MyStoredProcedure.ExecProc;  { does not return a result set }
Edit1.Text := MyStoredProcedure.ParamByName('OUTVAR').AsString;
MyStoredProcedure->ExecProc(); // Does not return a result set
Edit1->Text = MyStoredProcedure->ParamByName("OUTVAR")->AsString;
```

Note: *TADOStoredProc* does not have a *ParamByName* method. To obtain output parameter values when using ADO, access parameter objects using the *Parameters* property.

Tip: If you are executing the procedure multiple times, it is a good idea to set the *Prepared* property to *True*.

See Also

Fetching Multiple Result Sets (see page 1597)

Preparing Stored Procedures (see page 1607)

Working with Stored Procedure Parameters (see page 1625)

3.2.3.6.21 Fetching Multiple Result Sets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. If you are using *TSQLStoredProc* or *TADOStoredProc*, you can access the other sets of records by calling the *NextRecordSet* method:

```
var
  DataSet2: TCustomSQLDataSet;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
TCustomSQLDataSet *DataSet2 = SQLStoredProc1->NextRecordSet();
```

In *TSQLStoredProc, NextRecordSet* returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. In *TADOStoredProc*, *NextRecordset* returns an interface that can be assigned to the *RecordSet* property of an existing ADO dataset. For either class, the method returns the number of records in the returned dataset as an output parameter.

The first time you call *NextRecordSet*, it returns the second set of records. Calling *NextRecordSet* again returns a third dataset, and so on, until there are no more sets of records. When there are no additional cursors, *NextRecordSet* returns **nil**.

See Also

Executing Stored Procedures That Don't Return a Result Set (see page 1596)

Preparing Stored Procedures (see page 1607)

Working with Stored Procedure Parameters (see page 1625)

3.2.3.6.22 Limiting Records with Ranges

You can temporarily view and edit a subset of data for any dataset by using filters (see page 1631). Some table type datasets support an additional way to access a subset of available records, called ranges.

Ranges only apply to *TTable* and to client datasets. Despite their similarities, ranges and filters have different uses. The following topics discuss the differences between ranges and filters and how to use ranges:

- Understanding the differences between ranges and filters (see page 1615)
- Specifying ranges (see page 1610)
- Modifying a range (see page 1601)
- Applying or canceling a range (see page 1585)

See Also

Sorting Records with Indexes (see page 1609)

Using Indexes to Search for Records (see page 1615)

Creating Master/detail Relationships (see page 1590)

Controlling Read/write Access to Tables (see page 1587)

Creating and Deleting Tables (see page 1587)

Emptying Tables (see page 1592)

Synchronizing Tables (see page 1614)

3.2.3.6.23 Making the Table a Detail of Another Dataset

A table type dataset's *MasterSource* and *MasterFields* properties can be used to establish one-to-many relationships between two datasets.

The *MasterSource* property is used to specify a data source from which the table gets data from the master table. This data source can be linked to any type of dataset. For instance, by specifying a query's data source in this property, you can link a client dataset as the detail of the query, so that the client dataset tracks events occurring in the query.

The dataset is linked to the master table based on its current index. Before you specify the fields in the master dataset that are tracked by the detail dataset, first specify the index in the detail dataset that starts with the corresponding fields. You can use either the *IndexName* or the *IndexFieldNames* property.

Once you specify the index to use, use the *MasterFields* property to indicate the column(s) in the master dataset that correspond to the index fields in the detail table. To link datasets on multiple column names, separate field names with semicolons:

```
Parts.MasterFields := 'OrderNo;ItemNo';
Parts->MasterFields = "OrderNo;ItemNo";
```

To help create meaningful links between two datasets, you can use the Field Link designer. To use the Field Link designer, double click on the *MasterFields* property in the **Object Inspector** after you have assigned a *MasterSource* and an index.

The following steps create a simple form in which a user can scroll through customer records and display all orders for the current customer. The master table is the CustomersTable table, and the detail table is OrdersTable. The example uses the BDE-based TTable component, but you can use the same methods to link any table type datasets.

To create a simple form

1. Place two *TTable* components and two *TDataSource* components in a data module.
2. Set the properties of the following components,

Component	Property
First *TTable*	*DatabaseName*: DBDEMOS *TableName*: CUSTOMER *Name*: CustomersTable
Second *TTable*	*DatabaseName*: DBDEMOS *TableName*: ORDERS *Name*: OrdersTable
First *TDataSource*	*Name*: CustSource *DataSet*: CustomersTable
Second *TDataSource*	*Name*: OrdersSource *DataSet*: OrdersTable

3. Place two *TDBGrid* components on a form.
4. Choose **File** ▸ **Use Unit** to specify that the form should use the data module.
5. Set the DataSource property of the first grid component to "CustSource", and set the *DataSource* property of the second grid to "OrdersSource".
6. Set the MasterSource property of *OrdersTable* to "CustSource". This links the CUSTOMER table (the master table) to the ORDERS table (the detail table).
7. Double-click the MasterFields property value box in the **Object Inspector** to invoke the Field Link Designer to set the following properties:

- In the Available Indexes field, choose *CustNo* to link the two tables by the *CustNo* field.
- Select *CustNo* in both the Detail Fields and Master Fields field lists.
- Click the Add button to add this join condition. In the Joined Fields list,"CustNo -> CustNo" appears.

- Choose OK to commit your selections and exit the Field Link Designer.

8. Set the Active properties of *CustomersTable* and *OrdersTable* to *True* to display data in the grids on the form.
9. Compile and run the application.

If you run the application now, you will see that the tables are linked together, and that when you move to a new record in the CUSTOMER table, you see only those records in the ORDERS table that belong to the current customer.

See Also

Using Nested Detail Tables (see page 1616)

MasterSource

MasterSource

MasterSource

MasterSource

MasterSource

MasterSource

MasterFields

MasterFields

MasterFields

MasterFields

MasterFields

MasterFields

MasterFields

3.2.3.6.24 Marking and Returning to Records

In addition to moving from record to record in a dataset (or moving from one record to another by a specific number of records), it is often also useful to mark a particular location in a dataset so that you can return to it quickly when desired. TDataSet introduces a bookmarking feature that consists of a Bookmark property and five bookmark methods.

TDataSet implements **virtual** bookmark methods. While these methods ensure that any dataset object derived from *TDataSet* returns a value if a bookmark method is called, the return values are merely defaults that do not keep track of the current location. *TDataSet* descendants vary in the level of support they provide for bookmarks. None of the dbExpress datasets add any support for bookmarks. ADO datasets can support bookmarks, depending on the underlying database tables. BDE datasets, InterBase express datasets, and client datasets always support bookmarks.

The Bookmark property

The *Bookmark* property indicates which bookmark among any number of bookmarks in your application is current. *Bookmark* is a string that identifies the current bookmark. Each time you add another bookmark, it becomes the current bookmark.

The GetBookmark method

To create a bookmark, you must declare a variable of type *TBookmark* in your application, then call GetBookmark to allocate storage for the variable and set its value to a particular location in a dataset. The *TBookmark* type is a Pointer.

The GotoBookmark and BookmarkValid methods

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark. Before calling *GotoBookmark*, you can call *BookmarkValid to determine if the bookmark points to a record. BookmarkValid returns True*

if a specified bookmark points to a record.

The CompareBookmarks method

You can also call *CompareBookmarks to see if a bookmark you want to move to is different from another (or the current) bookmark. If the two bookmarks refer to the same record (or if both are* ***nil****), CompareBookmarks returns 0.*

The FreeBookmark method

FreeBookmark frees the memory allocated for a specified bookmark when you no longer need it. You should also call FreeBookmark before reusing an existing bookmark.

A bookmarking example

The following code illustrates one use of bookmarking:

```
procedure DoSomething (const Tbl: TTable)
var
  Bookmark: TBookmark;
begin
  Bookmark := Tbl.GetBookmark; { Allocate memory and assign a value }
  Tbl.DisableControls; { Turn off display of records in data controls }
  try
    Tbl.First; { Go to first record in table }
    while not Tbl.Eof do {Iterate through each record in table }
    begin
      { Do your processing here }
      .
      .
      .
      Tbl.Next;
    end;
  finally
    Tbl.GotoBookmark(Bookmark);
    Tbl.EnableControls; { Turn on display of records in data controls, if necessary }
    Tbl.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
  end;
end;
void DoSomething (const TTable *Tbl)
{
  TBookmark Bookmark = Tbl->GetBookmark(); // Allocate memory and assign a value
  Tbl->DisableControls(); // Turn off display of records in data controls
  try
  {
    for (Tbl->First(); !Tbl->Eof; Tbl->Next()) // Iterate through each record in table
    {
      // Do your processing here
      .
      .
      .
    }
  }
  __finally
  {
    Tbl->GotoBookmark(Bookmark);
    Tbl->EnableControls(); // Turn on display of records in data controls
    Tbl->FreeBookmark(Bookmark); // Deallocate memory for the bookmark
  }
}
```

Before iterating through records, controls are disabled. Should an error occur during iteration through records, the **finally** clause ensures that controls are always enabled and that the bookmark is always freed even if the loop terminates prematurely.

See Also

Using the First and Last Methods (see page 1623)

Using the Next and Prior Methods (see page 1624)

Using the MoveBy Method (see page 1623)

Using the Eof and Bof Properties (see page 1621)

3.2.3.6.25 Modifying a Range

Two functions enable you to modify the existing boundary conditions for a range: *EditRangeStart*, for changing the starting values for a range; and *EditRangeEnd*, for changing the ending values for the range.

To edit and apply a range

1. Putting the dataset into *dsSetKey* state and modifying the starting index value for the range.
2. Modifying the ending index value for the range.
3. Applying the range to the dataset.

You can modify either the starting or ending values of the range, or you can modify both boundary conditions. If you modify the boundary conditions for a range that is currently applied to the dataset, the changes you make are not applied until you call *ApplyRange* again.

Editing the start of a range

Call the *EditRangeStart* procedure to put the dataset into *dsSetKey* state and begin modifying the current list of starting values for the range. Once you call *EditRangeStart*, subsequent assignments to the Fields property overwrite the current index values to use when applying the range.

Tip: If you initially created a start range based on a partial key, you can use *EditRangeStart* to extend the starting value for a range.

Editing the end of a range

Call the *EditRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *EditRangeEnd*, subsequent assignments to the Fields property are treated as ending index values to use when applying the range.

See Also

Understanding the Differences Between Ranges and Filters (see page 1615)

Applying or Canceling a Range (see page 1585)

Setting Ranges (see page 1610)

EditRangeStart

EditRangeStart

EditRangeEnd

EditRangeEnd

ApplyRange

ApplyRange

3.2.3.6.26 Modifying Entire Records

On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record.

In code, however, you can use the following methods that work with entire record structures provided that the structure of the

database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records:

Methods that work with entire records

Method	Description
AppendRecord([array of values])	Appends a record with the specified column values at the end of a table; analogous to *Append*. Performs an implicit Post.
InsertRecord([array of values])	Inserts the specified values as a record before the current cursor position of a table; analogous to *Insert*. Performs an implicit *Post*.
SetFields([array of values])	Sets the values of the corresponding fields; analogous to assigning values to *TField*s. The application must perform an explicit *Post*.

These method take an array of values as an argument, where each value corresponds to a column in the underlying dataset. The values can be literals, variables, or NULL. If the number of values in an argument is less than the number of columns in a dataset, then the remaining values are assumed to be NULL.

For unindexed datasets, AppendRecord adds a record to the end of the dataset and InsertRecord inserts a record after the current cursor position. For indexed datasets, both methods place the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

SetFields assigns the values specified in the array of parameters to fields in the dataset. To use *SetFields*, an application must first call Edit to put the dataset in *dsEdit* mode. To apply the changes to the current record, it must perform a Post.

If you use *SetFields* to modify some, but not all fields in an existing record, you can pass NULL values for fields you do not want to change. If you do not supply enough values for all fields in a record, *SetFields* assigns NULL values to them. NULL values overwrite any existing values already in those fields.

For example, suppose a database has a COUNTRY table with columns for Name, Capital, Continent, Area, and Population. If a TTable component called *CountryTable* were linked to the COUNTRY table, the following statement would insert a record into the COUNTRY table:

```
CountryTable.InsertRecord(['Japan', 'Tokyo', 'Asia']);
CountryTable->InsertRecord(ARRAYOFCONST(("Japan", "Tokyo", "Asia")));
```

This statement does not specify values for Area and Population, so NULL values are inserted for them. The table is indexed on Name, so the statement would insert the record based on the alphabetic collation of "Japan".

To update the record, an application could use the following code:

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
TLocateOptions SearchOptions;
SearchOptions->Clear();
SearchOptions << loCaseInsensitive;
if (CountryTable->Locate("Name", "Japan", SearchOptions))
{
  CountryTable->Edit();
  CountryTable->SetFields(ARRAYOFCONST(((void *)NULL, (void *)NULL, (void *)NULL,
        344567, 164700000)));
  CountryTable->Post();
}
```

This code assigns values to the Area and Population fields and then posts them to the database. The three NULL pointers act as

place holders for the first three columns to preserve their current contents.

See Also

Editing Records (see page 1591)

Adding New Records (see page 1583)

Deleting Records (see page 1591)

Posting Data to the Database (see page 1606)

Canceling Changes (see page 1586)

3.2.3.6.27 Navigating Records in a Filtered Dataset

There are four dataset methods that navigate among records in a filtered dataset. The following table lists these methods and describes what they do:

Filtered dataset navigational methods

Method	Purpose
FindFirst	Move to the first record that matches the current filter criteria. The search for the first matching record always begins at the first record in the unfiltered dataset.
FindLast	Move to the last record that matches the current filter criteria.
FindNext	Moves from the current record in the filtered dataset to the next one.
FindPrior	Move from the current record in the filtered dataset to the previous one.

For example, the following statement finds the first filtered record in a dataset:

```
DataSet1.FindFirst;
DataSet1->FindFirst();
```

Provided that you set the Filter property or create an OnFilterRecord event handler for your application, these methods position the cursor on the specified record regardless of whether filtering is currently enabled. If you call these methods when filtering is not enabled, then they

- Temporarily enable filtering.
- Position the cursor on a matching record if one is found.
- Disable filtering.

  Note: If filtering is disabled and you do not set the *Filter* property or create an *OnFilterRecord* event handler, these methods do the same thing as *First*, *Last*, *Next*, and *Prior*.

  All navigational filter methods position the cursor on a matching record (if one is found), make that record the current one, and return *True*. If a matching record is not found, the cursor position is unchanged, and these methods return False. You can check the status of the Found property to wrap these calls, and only take action when *Found* is *True*. For example, if the cursor is already on the last matching record in the dataset and you call FindNext, the method returns *False*, and the current record is unchanged.

See Also

Enabling and Disabling Filtering (see page 1593)

Navigating Datasets (see page 1634)

3.2.3.6.28 Obtaining Information About Indexes

Your application can obtain information about server-defined indexes from all table type datasets. To obtain a list of available

indexes for the dataset, call the *GetIndexNames* method. *GetIndexNames* fills a string list with valid index names. For example, the following code fills a listbox with the names of all indexes defined for the *CustomersTable* dataset:

```
CustomersTable.GetIndexNames(ListBox1.Items);
CustomersTable->GetIndexNames(ListBox1->Items);
```

Note: For Paradox tables, the primary index is unnamed, and is therefore not returned by *GetIndexNames*. You can still change the index back to a primary index on a Paradox table after using an alternative index, however, by setting the *IndexName* property to a blank string.

To obtain information about the fields of the current index, use the

- *IndexFieldCount* property, to determine the number of columns in the index.
- *IndexFields* property, to examine a list the field components for the columns that comprise the index.

The following code illustrates how you might use *IndexFieldCount* and *IndexFields* to iterate through a list of column names in an application:

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20} of string;
begin
with CustomersTable do
  begin
  for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I].FieldName;
  end;
end;
AnsiString ListOfIndexFields[20];
for (int i = 0; i < CustomersTable->IndexFieldCount; i++)
  ListOfIndexFields[i] = CustomersTable->IndexFields[i]->FieldName;
```

Note: *IndexFieldCount* is not valid for a dBASE table opened on an expression index.

See Also

Specifying an Index with IndexName (see page 1630)

Creating an Index with IndexFieldNames (see page 1581)

GetIndexNames

GetIndexNames

GetIndexNames

GetIndexNames

GetIndexNames

GetIndexNames

GetIndexNames

3.2.3.6.29 Opening and Closing Datasets

To read or write data in a dataset, an application must first open it. You can open a dataset in two ways:

Open Method	Sample Code
Set theActiveproperty of the dataset to *True*, either at design time in the **Object Inspector**, or in code at runtime.	CustTable.Active := True; CustTable->Active = true;
Call the Open method for the dataset at runtime.	CustQuery.Open; CustQuery->Open();

When you open the dataset, the dataset first receives a BeforeOpen event, then it opens a cursor, populating itself with data, and finally, it receives an AfterOpen event.

The newly-opened dataset is in browse mode, which means your application can read the data and navigate through it.

You can close a dataset in two ways:

Close Method	Sample Code
Set the *Active* property of the dataset to *False*, either at design time in the **Object Inspector**, or in code at runtime.	CustQuery.Active := False; CustQuery->Active = false;
Call the Close method for the dataset at runtime.	CustTable.Close; CustTable->Close();

Just as the dataset receives *BeforeOpen* and *AfterOpen* events when you open it, it receives a BeforeClose and AfterClose event when you close it. You can use these events, for example, to prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
procedure TForm1.CustTableVerifyBeforeClose(DataSet: TDataSet);
begin
  if (CustTable.State in [dsEdit, dsInsert]) then begin
    case MessageDlg('Post changes before closing?', mtConfirmation, mbYesNoCancel, 0) of
      mrYes:    CustTable.Post;   { save the changes }
      mrNo:     CustTable.Cancel; { abandon the changes}
      mrCancel: Abort;            { abort closing the dataset }
    end;
  end;
end;
void __fastcall TForm1::VerifyBeforeClose(TDataSet *DataSet)
{
if (DataSet->State == dsEdit || DataSet->State == dsInsert)
{
TMsgDlgButtons btns;
btns << mbYes << mbNo;
    if (MessageDlg("Post changes before closing?", mtConfirmation, btns, 0) == mrYes)
DataSet->Post();
else
DataSet->Cancel();
}
}
```

Note: You may need to close a dataset when you want to change certain of its properties, such as TableName on a TTable component. When you reopen the dataset, the new property value takes effect.

See Also

Modifying Data (see page 1629)

Types of Datasets (see page 1635)

Determining Dataset States (see page 1633)

Navigating Datasets (see page 1634)

Searching Datasets (see page 1630)

Displaying and Editing a Subset of Data Using Filters (see page 1631)

Calculating Fields (see page 1585)

Using TDataSet Descendants (see page 1619)

3.2.3.6.30 Posting Data

After you finish editing a record, you must call the Post method to write out your changes. The *Post* method behaves differently, depending on the dataset's state and on whether you are caching updates.

- If you are not caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to the database and returns the dataset to the *dsBrowse* state.
- If you are caching updates, and the dataset is in the *dsEdit* or *dsInsert* state, *Post* writes the current record to an internal cache and returns the dataset to the *dsBrowse* state. The edits are not written to the database until you call *ApplyUpdates*.
- If the dataset is in the *dsSetKey* state, *Post* returns the dataset to the *dsBrowse* state.

Regardless of the initial state of the dataset, *Post* generates *BeforePost* and *AfterPost* events, before and after writing the current changes. You can use these events to update the user interface, or prevent the dataset from posting changes by calling the *Abort* procedure. If the call to *Post* fails, the dataset receives an *OnPostError* event, where you can inform the user of the problem or attempt to correct it.

Posting can be done explicitly, or implicitly as part of another procedure. When an application moves off the current record, *Post* is called implicitly. Calls to the First, Next, Prior, and Last methods perform a *Post* if the table is in *dsEdit* or *dsInsert* modes. The Append and Insert methods also implicitly post any pending data.

Warning: The Close method does not call *Post* implicitly. Use the BeforeClose event to post any pending edits explicitly.

See Also

Editing Records (see page 1591)

Adding New Records (see page 1583)

Deleting Records (see page 1591)

Canceling Changes (see page 1586)

Modifying Entire Records (see page 1601)

3.2.3.6.31 Preparing Queries

Preparing a query is an optional step that precedes query execution. Preparing a query submits the SQL statement and its parameters, if any, to the data access layer and the database server for parsing, resource allocation, and optimization. In some datasets, the dataset may perform additional setup operations when preparing the query. These operations improve query performance, making your application faster, especially when working with updatable queries.

An application can prepare a query by setting the *Prepared* property to *True*. If you do not prepare a query before executing it, the dataset automatically prepares it for you each time you call *Open* or *ExecSQL*. Even though the dataset prepares queries for you, you can improve performance by explicitly preparing the dataset before you open it the first time.

```
CustQuery.Prepared := True;
CustQuery->Prepared = true;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you add a parameter).

Note: When you change the text of the *SQL* property for a query, the dataset automatically closes and unprepares the query.

See Also

Specifying the Query (see page 1613)

Using Parameters in Queries (see page 1617)

Establishing Master/detail Relationships Using Parameters (see page 1593)

Executing Queries That Don't Return a Result Set (see page 1596)

Using Unidirectional Result Sets (see page 1624)

3.2.3.6.32 Preparing Stored Procedures

As with query-type datasets, stored procedure-type datasets must be prepared before they execute the stored procedure. Preparing a stored procedure tells the data access layer and the database server to allocate resources for the stored procedure and to bind parameters. These operations can improve performance.

If you attempt to execute a stored procedure before preparing it, the dataset automatically prepares it for you, and then unprepares it after it executes. If you plan to execute a stored procedure a number of times, it is more efficient to explicitly prepare it by setting the *Prepared* property to *True*.

```
MyProc.Prepared := True;
MyProc->Prepared = true;
```

When you explicitly prepare the dataset, the resources allocated for executing the stored procedure are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change the parameters when using Oracle overloaded procedures).

See Also

Fetching Multiple Result Sets (see page 1597)

Executing Stored Procedures that Don't Return a Result Set (see page 1596)

Working with Stored Procedure Parameters (see page 1625)

3.2.3.6.33 Searching On Partial Keys

Each time you call *SetKey* or *FindKey*, the method clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey* and *FindKey*.

For example, suppose you have already executed a search of the Employee table based on the City field of the "CityIndex" index. Suppose further that "CityIndex" includes both the *City* and *Company* fields. To find a record with a specified company name in a specified city, use the following code:

```
Employee.KeyFieldCount := 2;
Employee.EditKey;
Employee['Company'] := Edit2.Text;
Employee.GotoNearest;
Employee->KeyFieldCount = 2;
Employee->EditKey();
Employee->FieldValues["Company"] = Variant(Edit2->Text);
Employee->GotoNearest();
```

See Also

Executing a Search with Goto Methods (see page 1595)

Executing a Search with Find Methods (see page 1594)

Specifying an Index with IndexName (see page 1630)

Specifying the Current Record After a Successful Search (see page 1612)

Searching on Partial Keys (see page 1608)

3.2.3.6.34 Searching On Partial Keys

If the dataset has more than one key column, and you want to search for values in a subset of that key, set *KeyFieldCount* to the number of columns on which you are searching. For example, if the dataset's current index has three columns, and you want to search for values using just the first column, set *KeyFieldCount* to 1.

For table type datasets with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or the first, second, and third, but not just the first and third.

See Also

Executing a Search with Goto Methods (see page 1595)

Executing a Search with Find Methods (see page 1594)

Specifying the Current Record After a Successful Search (see page 1612)

Searching on Partial Keys (see page 1607)

3.2.3.6.35 Setting Filter Options

The FilterOptions property lets you specify whether a filter that compares string-based fields accepts records based on partial comparisons and whether string comparisons are case-sensitive. *FilterOptions* is a set property that can be an empty set (the default), or that can contain either or both of the following values:

FilterOptions values

Value	Meaning
foCaseInsensitive	Ignore case when comparing strings.
foNoPartialCompare	Disable partial string matching; that is, don't match strings that end with an asterisk (*).

For example, the following statements set up a filter that ignores case when comparing values in the *State* field:

```
FilterOptions := [foCaseInsensitive];
Filter := 'State = ' + QuotedStr('CA');
TFilterOptions FilterOptions;
FilterOptions->Clear();
FilterOptions << foCaseInsensitive;
Table1->FilterOptions = FilterOptions;
Table1->Filter = "State = "CA"";
```

See Also

Creating Filters (see page 1590)

3.2.3.6.36 Setting the Filter Property

To create a filter using the Filter property, set the value of the property to a string that contains the filter's test condition. For example, the following statement creates a filter that tests a dataset's *State* field to see if it contains a value for the state of California:

```
Dataset1.Filter := 'State = ' + QuotedStr('CA');
Dataset1->Filter = "State = 'CA'";
```

You can also supply a value for *Filter* based on text supplied by the user. For example, the following statement assigns the text in from edit box to *Filter*:

```
Dataset1.Filter := Edit1.Text;
Dataset1->Filter = Edit1->Text;
```

You can, of course, create a string based on both hard-coded text and user-supplied data:

```
Dataset1.Filter := 'State = ' + QuotedStr(Edit1.Text);
Dataset1->Filter = AnsiString("State = '") + Edit1->Text + "'";
```

Blank field values do not appear unless they are explicitly included in the filter:

```
Dataset1.Filter := 'State <> ''CA'' or State = BLANK';
Dataset1->Filter = "State <> 'CA' or State = BLANK";
```

Note: After you specify a value for *Filter*, to apply the filter to the dataset, set the Filtered property to *True*.

Filters can compare field values to literals and to constants using the following comparison and logical operators:

Comparison and logical operators that can appear in a filter

Operator	Meaning
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
AND	Tests two statements are both *True*
NOT	Tests that the following statement is not *True*
OR	Tests that at least one of two statements is *True*
+	Adds numbers, concatenates strings, adds numbers to date/time values (only available for some drivers)
-	Subtracts numbers, subtracts dates, or subtracts a number from a date (only available for some drivers)
*	Multiplies two numbers (only available for some drivers)
/	Divides two numbers (only available for some drivers)
*	wildcard for partial comparisons (*FilterOptions* must include *foPartialCompare*)

By using combinations of these operators, you can create fairly sophisticated filters. For example, the following statement checks to make sure that two test conditions are met before accepting a record for display:

```
(Custno > 1400) AND (Custno < 1500);
```

Note: When filtering is on, user edits to a record may mean that the record no longer meets a filter's test conditions. The next time the record is retrieved from the dataset, it may therefore "disappear." If that happens, the next record that passes the filter condition becomes the current record.

See Also

Writing an OnFilterRecord Event Handler (see page 1635)

3.2.3.6.37 Sorting Records with Indexes

An index determines the display order of records in a table. Typically, records appear in ascending order based on a primary, or default, index. This default behavior does not require application intervention. If you want a different sort order, however, you must specify either

- An alternate index.

- A list of columns on which to sort (not available on servers that aren't SQL-based).

Indexes let you present the data from a table in different orders. On SQL-based tables, this sort order is implemented by using the index to generate an ORDER BY clause in a query that fetches the table's records. On other tables (such as Paradox and dBASE tables), the index is used by the data access mechanism to present records in the desired order.

The following topics provide details on how to obtain information about available indexes and how to specify which index the dataset uses to sort records:

- Obtaining Information about Indexes (see page 1603)
- Specifying an Index with IndexName (see page 1630)
- Creating an Index with IndexFieldNames (see page 1581)

See Also

Using Indexes to Search for Records (see page 1615)

Limiting Records with Ranges (see page 1597)

Creating Master/detail Relationships (see page 1590)

Controlling Read/Write Access to Tables (see page 1587)

Creating and Deleting Tables (see page 1587)

Emptying Tables (see page 1592)

Synchronizing Tables (see page 1614)

3.2.3.6.38 Specifying Ranges

There are two mutually exclusive ways to specify a range:

- Specify the beginning and ending separately using *SetRangeStart* and *SetRangeEnd*.
- Specify both endpoints at once using *SetRange*.

Setting the beginning of a range

Call the *SetRangeStart* procedure to put the dataset into *dsSetKey* state and begin creating a list of starting values for the range. Once you call *SetRangeStart*, subsequent assignments to the Fields property are treated as starting index values to use when applying the range. Fields specified must apply to the current index.

For example, suppose your application uses a *TSimpleDataSet* component named *Customers*, linked to the CUSTOMER table, and that you have created persistent field components for each field in the *Customers* dataset. CUSTOMER is indexed on its first column (*CustNo*). A form in the application has two edit components named *StartVal* and *EndVal*, used to specify start and ending values for a range. The following code can be used to create and apply a range:

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo').AsString := StartVal.Text;
  SetRangeEnd;
  if (Length(EndVal.Text) > 0) then
    FieldByName('CustNo').AsString := EndVal.Text;
  ApplyRange;
end;
Customers->SetRangeStart();
Customers->FieldValues["CustNo"] = StrToInt(StartVal->Text);
Customers->SetRangeEnd();
if (!EndVal->Text.IsEmpty())
  Customers->FieldValues["CustNo"] = StrToInt(EndVal->Text);
Customers->ApplyRange();
```

This code checks that the text entered in *EndVal* is not null before assigning any values to *Fields*. If the text entered for *StartVal*

is null, then all records from the beginning of the dataset are included, since all values are greater than null. However, if the text entered for *EndVal* is null, then no records are included, since none are less than null.

For a multi-column index, you can specify a starting value for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. If you try to set a value for a field that is not in the index, the dataset raises an exception.

Tip: To start at the beginning of the dataset, omit the call to *SetRangeStart*.

To finish specifying the start of a range, call *SetRangeEnd* or apply or cancel the range (see page 1585).

Setting the end of a range

Call the *SetRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *SetRangeEnd*, subsequent assignments to the Fields property are treated as ending index values to use when applying the range. Fields specified must apply to the current index.

Warning: Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, Delphi assumes the ending value of the range is a null value. A range with null ending values is always empty.

The easiest way to assign ending values is to call the *FieldByName method. For example,*

```
with Contacts do
begin
  SetRangeStart;
  FieldByName('LastName').AsString := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName').AsString := Edit2.Text;
  ApplyRange;
end;
Contacts->SetRangeStart();
Contacts->FieldByName("LastName")->Value = Edit1->Text;
Contacts->SetRangeEnd();
Contacts->FieldByName("LastName")->Value = Edit2->Text;
Contacts->ApplyRange();
```

As with specifying start of range values, if you try to set a value for a field that is not in the index, the dataset raises an exception.

To finish specifying the end of a range, apply or cancel the range (see page 1585).

Setting start- and end-range values

Instead of using separate calls to *SetRangeStart* and *SetRangeEnd* to specify range boundaries, you can call the *SetRange* procedure to put the dataset into *dsSetKey* state and set the starting and ending values for a range with a single call.

SetRange takes two constant array parameters: a set of starting values, and a set of ending values. For example, the following statement establishes a range based on a two-column index:

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
TVarRec StartVals[2];
TVarRec EndVals[2];
StartVals[0] = Edit1->Text;
StartVals[1] = Edit2->Text;
EndVals[0] = Edit3->Text;
EndVals[1] = Edit4->Text;
Table1->SetRange(StartVals, 1, EndVals, 1);
```

For a multi-column index, you can specify starting and ending values for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. To omit a value for the first field in an index, and specify values for successive fields, pass a null value for the omitted field.

Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, the dataset assumes the ending value of the range is a null value. A range with null ending values is

always empty because the starting range is greater than or equal to the ending range.

Specifying a range based on partial keys

If a key is composed of one or more string fields, the *SetRange* methods support partial keys. For example, if an index is based on the *LastName* and *FirstName* columns, the following range specifications are valid:

```
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Zzzzzz';
Contacts.ApplyRange;
Contacts->SetRangeStart();
Contacts->FieldValues["LastName"] = "Smith";
Contacts->SetRangeEnd();
Contacts->FieldValues["LastName"] = "Zzzzzz";
Contacts->ApplyRange();
```

This code includes all records in a range where *LastName* is greater than or equal to "Smith." The value specification could also be:

```
Contacts['LastName'] := 'Sm';
Contacts->FieldValues["LastName"] = "Sm";
```

This statement includes records that have *LastName* greater than or equal to "Sm."

Including or excluding records that match boundary values

By default, a range includes all records that are greater than or equal to the specified starting range, and less than or equal to the specified ending range. This behavior is controlled by the *KeyExclusive* property. *KeyExclusive* is *False* by default.

If you prefer, you can set the *KeyExclusive* property for a dataset to *True* to exclude records equal to ending range. For example,

```
Contacts.KeyExclusive := True;
Contacts.SetRangeStart;
Contacts['LastName'] := 'Smith';
Contacts.SetRangeEnd;
Contacts['LastName'] := 'Tyler';
Contacts.ApplyRange;
Contacts->SetRangeStart();
Contacts->KeyExclusive = true;
Contacts->FieldValues["LastName"] = "Smith";
Contacts->SetRangeEnd();
Contacts->FieldValues["LastName"] = "Tyler";
Contacts->ApplyRange();
```

This code includes all records in a range where *LastName* is greater than or equal to "Smith" and less than "Tyler".

See Also

Understanding the Differences Between Ranges and Filters (see page 1615)

Applying or Canceling a Range (see page 1585)

Modifying a Range (see page 1601)

3.2.3.6.39 Specifying the Current Record After a Successful Search

By default, a successful search positions the cursor on the first record that matches the search criteria. If you prefer, you can set the *KeyExclusive* property to *True* to position the cursor on the next record after the first matching record.

By default, *KeyExclusive* is *False*, meaning that successful searches position the cursor on the first matching record.

See Also

Executing a Search with Goto Methods (see page 1595)

Executing a Search with Find Methods (see page 1594)

Searching on Partial Keys (see page 1608)

Searching on Partial Keys (see page 1607)

3.2.3.6.40 Specifying the Query

For true query-type datasets, you use the *SQL* property to specify the SQL statement for the dataset to execute. Some datasets, such as *TADODataSet*, *TSQLDataSet,* and client datasets, use a *CommandText* property to accomplish the same thing.

Most queries that return records are SELECT commands. Typically, they define the fields to include, the tables from which to select those fields, conditions that limit what records to include, and the order of the resulting dataset. For example:

```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = 1225
ORDER BY SaleDate
```

Queries that do not return records include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution. In most cases, the SQL command must be only one complete SQL statement, although that statement can be as complex as necessary (for example, a SELECT statement with a WHERE clause that uses several nested logical operators such as AND and OR). Some servers also support "batch" syntax that permits multiple statements; if your server supports such syntax, you can enter multiple statements when you specify the query.

The SQL statements used by queries can be verbatim, or they can contain replaceable parameters. Queries that use parameters are called *parameterized queries*. When you use parameterized queries, the actual values assigned to the parameters are inserted into the query before you execute, or run, the query. Using parameterized queries is very flexible, because you can change a user's view of and access to data on the fly at runtime without having to alter the SQL statement. For more information about parameterized queries, see Using parameters in queries (see page 1617).

Specifying a query using the SQL property

When using a true query-type dataset *(TQuery, TADOQuery, TSQLQuery,* or *TIBQuery)*, assign the query to the *SQL* property. The *SQL* property is a *TStrings* object. Each separate string in this *TStrings* object is a separate line of the query. Using multiple lines does not affect the way the query executes on the server, but can make it easier to modify and debug the query if you divide the statement into logical units:

```
MyQuery.Close;
MyQuery.SQL.Clear;
MyQuery.SQL.Add('SELECT CustNo, OrderNO, SaleDate');
MyQuery.SQL.Add(' FROM Orders');
MyQuery.SQL.Add('ORDER BY SaleDate');
MyQuery.Open;
MyQuery->Close();
MyQuery->SQL->Clear();
MyQuery->SQL->Add("SELECT CustNo, OrderNO, SaleDate");
MyQuery->SQL->Add("FROM Orders");
MyQuery->SQL->Add("ORDER BY SaleDate");
MyQuery->Open();
```

The code below demonstrates modifying only a single line in an existing SQL statement. In this case, the ORDER BY clause already exists on the third line of the statement. It is referenced via the *SQL* property using an index of 2.

```
MyQuery.SQL[2] := 'ORDER BY OrderNo';
MyQuery->SQL->Strings[2] = "ORDER BY OrderNO";
```

Note: The dataset must be closed when you specify or modify the SQL property.

At design time, use the String List editor to specify the query. Click the ellipsis button by the *SQL* property in the **Object Inspector** to display the String List editor.

Note: With some versions of Delphi, if you are using *TQuery*, you can also use the SQL Builder to construct a query based on a visible representation of tables and fields in a database. To use the SQL Builder, select the query component, right-click it to invoke the context menu, and choose Graphical Query Editor. To learn how to use SQL Builder, open it and use its online help.

Because the *SQL* property is a *TStrings* object, you can load the text of the query from a file by calling the *TStrings*.*LoadFromFile* method:

```
MyQuery.SQL.LoadFromFile('custquery.sql');
MyQuery->SQL->LoadFromFile("custquery.sql");
```

You can also use the *Assign* method of the *SQL* property to copy the contents of a string list object into the *SQL* property. The *Assign* method automatically clears the current contents of the *SQL* property before copying the new statement:

```
MyQuery.SQL.Assign(Memo1.Lines);
MyQuery->SQL->Assign(Memo1->Lines);
```

Specifying a query using the CommandText property

When using *TADODataSet*, *TSQLDataSet,* or a client dataset, assign the text of the query statement to the *CommandText* property:

```
MyQuery.CommandText := 'SELECT CustName, Address FROM Customer';
MyQuery->CommandText = "SELECT CustName, Address FROM Customer";
```

At design time, you can type the query directly into the **Object Inspector**, or, if the dataset already has an active connection to the database, you can click the ellipsis button by the *CommandText* property to display the Command Text editor. The Command Text editor lists the available tables, and the fields in those tables, to make it easier to compose your queries.

See Also

Using Parameters in Queries (see page 1617)

Establishing Master/detail Relationships Using Parameters (see page 1593)

Preparing Queries (see page 1606)

Executing Queries That Don't Return a Result Set (see page 1596)

Using Unidirectional Result Sets (see page 1624)

3.2.3.6.41 Synchronizing Tables

If you have two or more datasets that represent the same database table but do not share a data source component, then each dataset has its own view on the data and its own current record. As users access records through each datasets, the components' current records will differ.

If the datasets are all instances of *TTable*, or all instances of *TIBTable*, or all client datasets, you can force the current record for each of these datasets to be the same by calling the *GotoCurrent* method. *GotoCurrent* sets its own dataset's current record to the current record of a matching dataset. For example, the following code sets the current record of *CustomerTableOne* to be the same as the current record of *CustomerTableTwo*:

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
CustomerTableOne->GotoCurrent(CustomerTableTwo);
```

Tip: If your application needs to synchronize datasets in this manner, put the datasets in a data module and add the unit for the data module to the uses clause of each unit that accesses the tables.

To synchronize datasets from separate forms, you must add one form's unit to the uses clause of the other, and you must

qualify at least one of the dataset names with its form name. For example:

```
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
CustomerTableOne->GotoCurrent(Form2->CustomerTableTwo);
```

See Also

Sorting Records with Indexes (see page 1609)

Limiting Records with Ranges (see page 1597)

Using Indexes to Search for Records (see page 1615)

Creating Master/detail Relationships (see page 1590)

Controlling Read/write Access to Tables (see page 1587)

Creating and Deleting Tables (see page 1587)

Emptying Tables (see page 1592)

3.2.3.6.42 Understanding the Differences Between Ranges and Filters

Both ranges and filters restrict visible records to a subset of all available records, but the way they do so differs. A range is a set of contiguously indexed records that fall between specified boundary values. For example, in an employee database indexed on last name, you might apply a range to display all employees whose last names are greater than "Jones" and less than "Smith". Because ranges depend on indexes, you must set the current index to one that can be used to define the range. As with specifying an index to sort records (see page 1609), you can assign the index on which to define a range using either the *IndexName* or the *IndexFieldNames* property.

A filter, on the other hand, is any set of records that share specified data points, regardless of indexing. For example, you might filter an employee database to display all employees who live in California and who have worked for the company for five or more years. While filters can make use of indexes if they apply, filters are not dependent on them. Filters are applied record-by-record as an application scrolls through a dataset.

In general, filters are more flexible than ranges. Ranges, however, can be more efficient when datasets are large and the records of interest to an application are already blocked in contiguously indexed groups. For very large datasets, it may be still more efficient to use the WHERE clause of a query-type dataset to select data. For details on specifying a query, see Using query-type datasets (see page 1618).

See Also

Specifying Ranges (see page 1610)

Applying or Canceling a Range (see page 1585)

Modifying a Range (see page 1601)

Displaying and Editing a Subset of Data Using Filters (see page 1631)

3.2.3.6.43 Using Indexes to Search for Records

You can search against any dataset using the Locate and Lookup methods of *TDataSet*. However, by explicitly using indexes, some table type datasets can improve over the searching performance provided by the *Locate* and *Lookup* methods.

ADO datasets all support the Seek method, which moves to a record based on a set of field values for fields in the current index. *Seek* lets you specify where to move the cursor relative to the first or last matching record.

TTable and all types of client dataset support similar indexed-based searches, but use a combination of related methods. The following table summarizes the six related methods provided by *TTable* and client datasets to support index-based searches:

Index-based search methods

Method	Purpose
EditKey	Preserves the current contents of the search key buffer and puts the dataset into *dsSetKey* state so your application can modify existing search criteria prior to executing a search.
FindKey	Combines the *SetKey* and *GotoKey* methods in a single method.
FindNearest	Combines the *SetKey* and *GotoNearest* methods in a single method.
GotoKey	Searches for the first record in a dataset that exactly matches the search criteria, and moves the cursor to that record if one is found.
GotoNearest	Searches on string-based fields for the closest match to a record based on partial key values, and moves the cursor to that record.
SetKey	Clears the search key buffer and puts the table into *dsSetKey* state so your application can specify new search criteria prior to executing a search.

GotoKey and *FindKey* are boolean functions that, if successful, move the cursor to a matching record and return *True*. If the search is unsuccessful, the cursor is not moved, and these functions return *False*.

GotoNearest and *FindNearest* always reposition the cursor either on the first exact match found or, if no match is found, on the first record that is greater than the specified search criteria.

The following topics discuss the Goto and Find methods in greater detail:

- Executing a Search with Goto Methods (see page 1595)
- Executing a Search with Find Methods (see page 1594)
- Specifying the Current Record After a Successful Search (see page 1612)
- Searching on Partial Keys (see page 1608)
- Repeating or Extending a Search (see page 1607)

See Also

Sorting Records with Indexes (see page 1609)

Limiting Records with Ranges (see page 1597)

Creating Master/detail Relationships (see page 1590)

Controlling Read/Write Access to Tables (see page 1587)

Creating and Deleting Tables (see page 1587)

Emptying Tables (see page 1592)

Synchronizing Tables (see page 1614)

3.2.3.6.44 Using Nested Detail Tables

A nested table is a detail dataset that is the value of a single dataset field in another (master) dataset. For datasets that represent server data, a nested detail dataset can only be used for a dataset field on the server. *TClientDataSet* components do not represent server data, but they can also contain dataset fields if you create a dataset for them that contains nested details, or if they receive data from a provider that is linked to the master table of a master/detail relationship.

Note: For *TClientDataSet*, using nested detail sets is necessary if you want to apply updates from master and detail tables to a database server.

To use nested detail sets, the ObjectView property of the master dataset must be *True*. When your table type dataset contains nested detail datasets, *TDBGrid* provides support for displaying the nested details in a popup window. For more information on

how this works, see Working with dataset fields (see page 1867).

Alternately, you can display and edit detail datasets in data-aware controls by using a separate dataset component for the detail set. At design time, create persistent fields for the fields in your (master) dataset, using the Fields Editor: right click the master dataset and choose Fields Editor. Add a new persistent field to your dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of the detail table. You must also add persistent fields for any other fields used in your master dataset.

The dataset component for the detail table is a dataset descendant of a type allowed by the master table. *TTable* components only allow *TNestedDataSet* components as nested datasets. *TSQLTable* components allow other *TSQLTable* components. *TClientDataset* components allow other client datasets. Choose a dataset of the appropriate type from the **Tool palette** and add it to your form or data module. Set this detail dataset's DataSetField property to the persistent DataSet field in the master dataset. Finally, place a data source component on the form or data module and set its *DataSet* property to the detail dataset. Data-aware controls can use this data source to access the data in the detail set.

See Also

Making the Table a Detail of Another Dataset (see page 1598)

3.2.3.6.45 Using Parameters in Queries

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values, such as those used in a WHERE clause for comparisons, that appear in an SQL statement. Ordinarily, parameters stand in for data values passed to the statement. For example, in the following INSERT statement, values to insert are passed as parameters:

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

In this SQL statement, *:Name*, *:Capital*, and *:Population* are placeholders for actual values supplied to the statement at runtime by your application. Note that the names of parameters begin with a colon. The colon is required so that the parameter names can be distinguished from literal values. You can also include unnamed parameters by adding a question mark (?) to your query. Unnamed parameters are identified by position, because they do not have unique names.

Before the dataset can execute the query, you must supply values for any parameters in the query text. *TQuery*, *TIBQuery*, *TSQLQuery*, and client datasets use the *P*arams property to store these values. *TADOQuery* uses the *Parameters* property instead. *Params* (or *Parame*ters) is a collection of parameter objects (TParam or TParameter), where each object represents a single parameter. When you specify the text for the query, the dataset generates this set of parameter objects, and (depending on the dataset type) initializes any of their properties that it can deduce from the query.

Note: You can suppress the automatic generation of parameter objects in response to changing the query text by setting the *ParamCheck* property to *False*. This is useful for data definition language (DDL) statements that contain parameters as part of the DDL statement that are not parameters for the query itself. For example, the DDL statement to create a stored procedure may define parameters that are part of the stored procedure. By setting *ParamCheck* to *False*, you prevent these parameters from being mistaken for parameters of the query.

Parameter values must be bound into the SQL statement before it is executed for the first time. Query components do this automatically for you even if you do not explicitly call the *Prepare* method before executing a query.

Tip: It is a good programming practice to provide variable names for parameters that correspond to the actual name of the column with which it is associated. For example, if a column name is "Number," then its corresponding parameter would be ":Number". Using matching names is especially important if the dataset uses a data source to obtain parameter values from another dataset. This process is described in Establishing master/detail relationships using parameters (see page 1593).

The following topics describe how to specify the datatypes and values of parameters for your query:

- Supplying Parameters at Design Time (see page 1582)

- Supplying Parameters at Runtime (see page 1583)

See Also

Specifying the Query (see page 1613)

Establishing Master/detail Relationships Using Parameters (see page 1593)

Preparing Queries (see page 1606)

Executing Queries That Don't Return a Result Set (see page 1596)

Using Unidirectional Result Sets (see page 1624)

3.2.3.6.46 Using Query-type Datasets

To use a query-type dataset

1. Place the appropriate dataset component in a data module or on a form, and set its Name property to a unique value appropriate to your application.
2. Identify the database server to query. Each query-type dataset does this differently, but typically you specify a database connection component:

- For *TQuery*, specify a *TDatabase* component or a BDE alias using the DatabaseName property.
- For *TADOQuery*, specify a *TADOConnection* component using the Connection property.
- For *TSQLQuery*, specify a *TSQLConnection* component using the SQLConnection property.
- For *TIBQuery*, specify a *TIBConnection* component using the Database property. For information about using database connection components, see Connecting to databases (see page 1506)

3. Specify an SQL statement (see page 1613) in the *SQL* property of the dataset, and optionally specify any parameters (see page 1617) for the statement.
4. If the query data is to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the query-type dataset. The data source component forwards the results of the query (called a *result set*) to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
5. Activate the query component. For queries that return a result set, use the *Active* property or the *Open* method. To execute queries that only perform an action on a table and return no result set (see page 1596), use the *ExecSQL* method at runtime. If you plan to execute the query more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the query. For information about preparing a query, see Preparing queries (see page 1606).

In addition to the basic steps described above, the following topics describe how to establish master/detsil relationships when using query-type datasets and how to improve performance when you only need a unidirectional cursor:

- Establishing master/detail relationships using parameters (see page 1593)
- Using unidirectional result sets (see page 1624)

See Also

Using Table Type Datasets (see page 1620)

Using Stored Procedure-type Datasets (see page 1618)

3.2.3.6.47 Using Stored Procedure-type Datasets

How your application uses a stored procedure depends on how the stored procedure was coded, whether and how it returns data, the specific database server used, or a combination of these factors.

To access a stored procedure on a server

1. Place the appropriate dataset component in a data module or on a form, and set its Name property to a unique value appropriate to your application.
2. Identify the database server that defines the stored procedure. Each stored procedure-type dataset does this differently, but typically you specify a database connection component:

- For *TStoredProc*, specify a *TDatabase* component or a BDE alias using the DatabaseName property.
- For *TADOStoredProc*, specify a *TADOConnection* component using the Connection property.
- For *TSQLStoredProc*, specify a *TSQLConnection* component using the SQLConnection property.
- For *TIBStoredProc*, specify a *TIBConnection* component using the Database property. For information about using database connection components, see Connecting to databases (see page 1506)

3. Specify the stored procedure to execute. For most stored procedure-type datasets, you do this by setting the *StoredProcName* property. The one exception is *TADOStoredProc*, which has a *ProcedureName* property instead.
4. If the stored procedure returns a cursor to be used with visual data controls, add a data source component to the data module, and set its *DataSet* property to the stored procedure-type dataset. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
5. Provide input parameter values for the stored procedure, if necessary. If the server does not provide information about all stored procedure parameters, you may need to provide additional input parameter information, such as parameter names and data types. For information about working with stored procedure parameters, see Working with stored procedure parameters (see page 1625).
6. Execute the stored procedure. For stored procedures that return a cursor, use the *Active* property or the *Open* method. To execute stored procedures that do not return any results or that only return output parameters (see page 1596), use the *ExecProc* method at runtime. If you plan to execute the stored procedure more than once, you may want to call *Prepare* to initialize the data access layer and bind parameter values into the stored procedure. For information about preparing a query, see Preparing stored procedures (see page 1607).
7. Process any results. These results can be returned as result and output parameters, or they can be returned as a result set that populates the stored procedure-type dataset. Some stored procedures return multiple cursors. For details on how to access the additional cursors, see Fetching multiple result sets (see page 1597).

See Also

Using Table Type Datasets (see page 1620)

Using Query-type Datasets (see page 1618)

3.2.3.6.48 Using TDataSet Descendants

TDataSet has several immediate descendants, each of which corresponds to a different data access mechanism. You do not work directly with any of these descendants. Rather, each descendant introduces the properties and methods for using a particular data access mechanism. These properties and methods are then exposed by descendant classes that are adapted to different types of server data. The immediate descendants of *TDataSet* include

- *TBDEDataSet*, which uses the Borland Database Engine (BDE) to communicate with the database server. The *TBDEDataSet* descendants you use are *TTable*, *TQuery*, *TStoredProc*, and *TNestedTable*. The unique features of BDE-enabled datasets are described in Using the Borland Database Engine (see page 1643)
- *TCustomADODataSet*, which uses ActiveX Data Objects (ADO) to communicate with an OLEDB data store. The *TCustomADODataSet* descendants you use are *TADODataSet*, *TADOTable*, *TADOQuery*, and *TADOStoredProc*. The unique features of ADO-based datasets are described in Working with ADO components. (see page 1494)
- *TCustomSQLDataSet*, which uses dbExpress to communicate with a database server. The *TCustomSQLDataSet* descendants you use are *TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*. The unique features of dbExpress datasets are described in Using Unidirectional Datasets. (see page 1823)
- *TIBCustomDataSet*, which communicates directly with an InterBase database server. The *TIBCustomDataSet* descendants

you use are *TIBDataSet*, *TIBTable*, *TIBQuery*, and *TIBStoredProc*.

- *TCustomClientDataSet*, which represents the data from another dataset component or the data from a dedicated file on disk. The *TCustomClientDataSet* descendants you use are *TClientDataSet*, which can connect to an external (source) dataset, and the client datasets that are specialized to a particular data access mechanism (*TBDEClientDataSet,TSimpleDataSet*, and *TIBClientDataSet*), which use an internal source dataset. The unique features of client datasets are described in Using client datasets (see page 1740)

Some pros and cons of the various data access mechanisms employed by these *TDataSet* descendants are described in Using databases (see page 1572).

In addition to the built-in datasets, you can create your own custom *TDataSet* descendants—for example to supply data from a process other than a database server, such as a spreadsheet. Writing custom datasets allows you the flexibility of managing the data using any method you choose, while still letting you use the VCL data controls to build your user interface. For more information about creating custom components, see Overview of component creation. (see page 1313)

Although each *TDataSet* descendant has its own unique properties and methods, some of the properties and methods introduced by descendant classes are the same as those introduced by other descendant classes that use another data access mechanism. For example, there are similarities between the "table" components (*TTable*, *TADOTable*, *TSQLTable*, and *TIBTable*). For information about the commonalities introduced by *TDataSet* descendants, see Types of datasets (see page 1635).

See Also

Opening and Closing Datasets (see page 1604)

Determining Dataset States (see page 1633)

Navigating Datasets (see page 1634)

Searching Datasets (see page 1630)

Displaying and Editing a Subset of Data Using Filters (see page 1631)

Modifying Data (see page 1629)

Database Architecture (see page 1562)

Calculating Fields (see page 1585)

3.2.3.6.49 Using Table Type Datasets

To use a table type dataset

1. Place the appropriate dataset component in a data module or on a form, and set its Name property to a unique value appropriate to your application.
2. Identify the database server that contains the table you want to use. Each table type dataset does this differently, but typically you specify a database connection component:

- For *TTable*, specify a *TDatabase* component or a BDE alias using the DatabaseName property.
- For *TADOTable*, specify a *TADOConnection* component using the Connection property.
- For *TSQLTable*, specify a *TSQLConnection* component using the SQLConnection property.
- For *TIBTable*, specify a *TIBConnection* component using the Database property. For information about using database connection components, see Connecting to databases (see page 1506)

3. Set the *TableName* property to the name of the table in the database. You can select tables from a drop-down list if you have already identified a database connection component.
4. Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the dataset. The data source component is used to pass a result set from the dataset to data-aware components for display.

Advantages of using table type datasets

The main advantage of using table type datasets is the availability of indexes. Indexes enable your application to

- Sort the Records in the Dataset (see page 1609).
- Locate Records Quickly (see page 1615).
- Limit the Records That are Visible (see page 1597).
- Establish Master/Detail Relationships (see page 1590).

In addition, the one-to-one relationship between table type datasets and database tables enables many of them to be used for

- Controlling Read/Write Access To Tables (see page 1587)
- Creating and Deleting Tables (see page 1587)
- Emptying Tables (see page 1592)
- Synchronizing Tables (see page 1614)

See Also

Using Query-type Datasets (see page 1618)

Using Stored Procedure-type Datasets (see page 1618)

3.2.3.6.50 Using the Eof and Bof Properties

Two read-only, runtime properties, *Eof* (End-of-file) and *Bof(Beginning-of-file), are useful when you want to iterate through all records in a dataset.*

Eof

When EOF is *True*, it indicates that the cursor is unequivocally at the last row in a dataset. *Eof* is set to *True* when an application

- Opens an empty dataset.
- Calls a dataset's *Last* method.
- Calls a dataset's *Next* method, and the method fails (because the cursor is currently at the last row in the dataset.
- Calls *SetRange* on an empty range or dataset.

Eof is set to *False* in all other cases; you should assume *Eof* is *False* unless one of the conditions above is met *and* you test the property directly.

Eof is commonly tested in a loop condition to control iterative processing of all records in a dataset. If you open a dataset containing records (or you call First) *Eof* is *False*. To iterate through the dataset a record at a time, create a loop that steps through each record by calling Next, and terminates when *Eof* is *True*. *Eof* remains *False* until you call *Next* when the cursor is already on the last record.

The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls;
try
  CustTable.First; { Go to first record, which sets Eof False }
  while not CustTable.Eof do { Cycle until Eof is True }
  begin
    { Process each record here }
    .
    .
    .
    CustTable.Next; { Eof False on success; Eof True when Next fails on last record }
  end;
finally
  CustTable.EnableControls;
end;
CustTable->DisableControls(); // Speed up processing; prevent screen flicker
try
{
  while (!CustTable->Bof) // Cycle until Bof is true
  (
```

```
    // Process each record here
    .
    .
    .
    CustTable->Prior();
    // Bof false on success; Bof true when Prior fails on first record
  }
}
__finally
{
  CustTable->EnableControls();
}
CustTable->DisableControls();
try
{
  for (CustTable->First(); !CustTable->Eof; CustTable->Next())
  (
    // Process each record here
    .
    .
    .
  }
}
__finally
{
  CustTable->EnableControls();
}
```

Tip: This example also shows how to disable and enable data-aware visual controls tied to a dataset. If you disable visual controls during dataset iteration, it speeds processing because your application does not need to update the contents of the controls as the current record changes. After iteration is complete, controls should be enabled again to update them with values for the new current row. Note that enabling of the visual controls takes place in the **finally** clause of a **try...finally** statement. This guarantees that even if an exception terminates loop processing prematurely, controls are not left disabled.

Bof

When BOF is *True*, it indicates that the cursor is unequivocally at the first row in a dataset. *Bof* is set to *True* when an application

- Opens a dataset.
- Calls a dataset's First method.
- Calls a dataset's Prior method, and the method fails (because the cursor is currently at the first row in the dataset.
- Calls *SetRange* on an empty range or dataset.

Bof is set to *False* in all other cases; you should assume *Bof* is *False* unless one of the conditions above is met *and* you test the property directly.

Like EOF, *Bof* can be in a loop condition to control iterative processing of records in a dataset. The following code illustrates one way you might code a record-processing loop for a dataset called *CustTable*:

```
CustTable.DisableControls; { Speed up processing; prevent screen flicker }
try
  while not CustTable.Bof do { Cycle until Bof is True }
  begin
    { Process each record here }
    .
    .
    .
    CustTable.Prior; { Bof False on success; Bof True when Prior fails on first record }
  end;
finally
  CustTable.EnableControls; { Display new current row in controls }
end;
```

See Also

Using the First and Last Methods (see page 1623)

Using the Next and Prior Methods (see page 1624)

Using the MoveBy Method (see page 1623)

Marking and Returning to Records (see page 1599)

3.2.3.6.51 Using the First and Last Methods

The First method moves the cursor to the first row in a dataset and sets the BOF property to *True*. If the cursor is already at the first row in the dataset, *First* does nothing.

For example, the following code moves to the first record in *CustTable*:

```
CustTable.First;
CustTable->First();
```

The Last method moves the cursor to the last row in a dataset and sets the EOF property to *True*. If the cursor is already at the last row in the dataset, *Last* does nothing.

The following code moves to the last record in *CustTable*:

```
CustTable.Last;
CustTable->Last();
```

Note: The *Last* method raises an exception in unidirectional datasets.

Tip: While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you can also enable your users to navigate from record to record using the TDBNavigator component. The navigator component contains buttons that, when active and visible, enable a user to move to the first and last rows of an active dataset. The OnClick events for these buttons call the *First* and *Last* methods of the dataset. For more information about making effective use of the navigator component, see Navigating and manipulating records (see page 1767).

See Also

Using the Next and Prior Methods (see page 1624)

Using the MoveBy Method (see page 1623)

Using the Eof and Bof Properties (see page 1621)

Marking and Returning to Records (see page 1599)

3.2.3.6.52 Using the MoveBy Method

MoveBy lets you specify how many rows forward or back to move the cursor in a dataset. Movement is relative to the current record at the time that *MoveBy* is called. *MoveBy* also sets the BOF and EOF properties for the dataset as appropriate.

This function takes an integer parameter, the number of records to move. Positive integers indicate a forward move and negative integers indicate a backward move.

Note: *MoveBy* raises an exception in unidirectional datasets if you use a negative argument.

MoveBy returns the number of rows it moves. If you attempt to move past the beginning or end of the dataset, the number of rows returned by *MoveBy* differs from the number of rows you requested to move. This is because *MoveBy* stops when it reaches the first or last record in the dataset.

The following code moves two records backward in *CustTable*:

```
CustTable.MoveBy(-2);
CustTable->MoveBy(-2);
```

Note: If your application uses *MoveBy* in a multi-user database environment, keep in mind that datasets are fluid. A record that was five records back a moment ago may now be four, six, or even an unknown number of records back if several users are simultaneously accessing the database and changing its data.

See Also

Using the First and Last Methods (see page 1623)

Using the Next and Prior Methods (see page 1624)

Using the Eof and Bof Properties (see page 1621)

Marking and Returning to Records (see page 1599)

3.2.3.6.53 Using the Next and Prior Methods

The Next method moves the cursor forward one row in the dataset and sets the BOF property to *False* if the dataset is not empty. If the cursor is already at the last row in the dataset when you call *Next*, nothing happens.

For example, the following code moves to the next record in *CustTable*:

```
CustTable.Next;
CustTable->Next();
```

The Prior method moves the cursor back one row in the dataset, and sets*EOF to False if the dataset is not empty. If the cursor is already at the first row in the dataset when you call Prior, Prior does nothing.*

For example, the following code moves to the previous record in CustTable:

```
CustTable.Prior;
CustTable->Prior();
```

Note: The *Prior* method raises an exception in unidirectional datasets.

See Also

Using the First and Last Methods (see page 1623)

Using the MoveBy Method (see page 1623)

Using the Eof and Bof Properties (see page 1621)

Marking and Returning to Records (see page 1599)

3.2.3.6.54 Using Unidirectional Result Sets

When a query-type dataset returns a result set, it also receives a cursor, or pointer to the first record in that result set. The record pointed to by the cursor is the currently active record. The current record is the one whose field values are displayed in data-aware components associated with the result set's data source. Unless you are using dbExpress, this cursor is bi-directional by default. A bi-directional cursor can navigate both forward and backward through its records. Bi-directional cursor support requires some additional processing overhead, and can slow some queries.

If you do not need to be able to navigate backward through a result set, *TQuery* and *TIBQuery* let you improve query performance by requesting a unidirectional cursor instead. To request a unidirectional cursor, set the *UniDirectional* property to True.

Set *UniDirectional* before preparing and executing a query. The following code illustrates setting *UniDirectional* prior to preparing

and executing a query:

```
if not (CustomerQuery.Prepared) then
begin
  CustomerQuery.UniDirectional := True;
  CustomerQuery.Prepared := True;
end;
CustomerQuery.Open;  { returns a result set with a one-way cursor }
```

```
if (!CustomerQuery->Prepared)
{
  CustomerQuery->UniDirectional = true;
CustomerQuery->Prepared = true;
}
CustomerQuery->Open(); // Returns a result set with a one-way cursor
```

Note: Do not confuse the *UniDirectional* property with a unidirectional dataset. Unidirectional datasets (*TSQLDataSet*, *TSQLTable*, *TSQLQuery*, and *TSQLStoredProc*) use dbExpress, which only returns unidirectional cursors. In addition to restricting the ability to navigate backwards, unidirectional datasets do not buffer records, and so have additional limitations (such as the inability to use filters).

See Also

Specifying the Query (see page 1613)

Using Parameters in Queries (see page 1617)

Establishing Master/detail Relationships Using Parameters (see page 1593)

Preparing Queries (see page 1606)

Executing Queries That Don't Return a Result Set (see page 1596)

3.2.3.6.55 Working with Stored Procedure Parameters

There are four types of parameters that can be associated with stored procedures:

- *Input p*arameters, used to pass values to a stored procedure for processing.
- *Output* parameters, used by a stored procedure to pass return values to an application.
- *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application.
- *A result parameter*, used by some stored procedures to return an error or status value to an application. A stored procedure can only return one result parameter.

Whether a stored procedure uses a particular type of parameter depends both on the general language implementation of stored procedures on your database server and on a specific instance of a stored procedure. For any server, individual stored procedures may or may not use input parameters. On the other hand, some uses of parameters are server-specific. For example, on MS-SQL Server and Sybase stored procedures always return a result parameter, but the InterBase implementation of a stored procedure never returns a result parameter.

Access to stored procedure parameters is provided by the *Params* property (in *TStoredProc*, *TSQLStoredProc*, *TIBStoredProc*) or the *Parameters* property (in *TADOStoredProc*). When you assign a value to the *StoredProcName* (or *ProcedureName*) property, the dataset automatically generates objects for each parameter of the stored procedure. For some datasets, if the stored procedure name is not specified until runtime, objects for each parameter must be programmatically created at that time. Not specifying the stored procedure and manually creating the *TParam* or *TParameter* objects allows a single dataset to be used with any number of available stored procedures.

Note: Some stored procedures return a dataset in addition to output and result parameters. Applications can display dataset records in data-aware controls, but must separately process output and result parameters.

Setting up parameters at design time

You can specify stored procedure parameter values at design time using the parameter collection editor. To display the parameter collection editor, click on the ellipsis button for the *Params* or *Parameters* property in the **Object Inspector**.

Warning: You can assign values to input parameters by selecting them in the parameter collection editor and using the Object Inspector

to set the *Value* property. However, do not change the names or data types for input parameters reported by the server. Otherwise, when you execute the stored procedure an exception is raised. Some servers do not report parameter names or data types. In these cases, you must set up the parameters manually using the parameter collection editor. Right click and choose Add to add parameters. For each parameter you add, you must fully describe the parameter. Even if you do not need to add any parameters, you should check the properties of individual parameter objects to ensure that they are correct.

If the dataset has a *Params* property (*TParam* objects), the following properties must be correctly specified:

- The Name property indicates the name of the parameter as it is defined by the stored procedure.
- The DataType property gives the data type for the parameter's value. When using *TSQLStoredProc*, some data types require additional information:
- The NumericScale property indicates the number of decimal places for numeric parameters.
- The Precision property indicates the total number of digits for numeric parameters.
- The Size property indicates the number of characters in string parameters.
- The ParamType property indicates the type of the selected parameter. This can be *ptInput* (for input parameters), *ptOutput* (for output parameters), *ptInputOutput* (for input/output parameters) or *ptResult* (for result parameters).
- The Value property specifies a value for the selected parameter. You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.

If the dataset uses a *Parameters* property (*TParameter* objects), the following properties must be correctly specified:

- The Name property indicates the name of the parameter as it is defined by the stored procedure.
- The DataType property gives the data type for the parameter's value. For some data types, you must provide additional information:
- The NumericScale property indicates the number of decimal places for numeric parameters.
- The Precision property indicates the total number of digits for numeric parameters.
- The Size property indicates the number of characters in string parameters.
- The Direction property gives the type of the selected parameter. This can be *pdInput* (for input parameters), *pdOutput* (for output parameters), *pdInputOutput* (for input/output parameters) or *pdReturnValue* (for result parameters).
- The Attributes property controls the type of values the parameter will accept. *Attributes* may be set to a combination of *psSigned*, *psNullable*, and *psLong*.
- The Value property specifies a value for the selected parameter. Do not set values for output and result parameters. For input and input/output parameters, you can leave *Value* blank if your application supplies parameter values at runtime.

Using parameters at runtime

With some datasets, if the name of the stored procedure is not specified until runtime, no *TParam* objects are automatically created for parameters and they must be created programmatically. This can be done using the *TParam.Create* method or the *TParams.AddParam* method:

```
var
  P1, P2: TParam;
begin
  ...
  with StoredProc1 do begin
    StoredProcName := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := 'EMP_NO';
      Params[1].Name := 'PROJ_ID';
```

```
      ParamByname('EMP_NO').AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname('PROJ_ID').AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
  ...
end;
TParam *P1, *P2;
StoredProc1->StoredProcName = "GET_EMP_PROJ";
StoredProc1->Params->Clear();
P1 = new TParam(StoredProc1->Params, ptInput);
P2 = new TParam(StoredProc1->Params, ptOutput);
try
{
  StoredProc1->Params->Items[0]->Name = "EMP_NO";
  StoredProc1->Params->Items[1]->Name = "PROJ_ID";
  StoredProc1->ParamByName("EMP_NO")->AsSmallInt = 52;
  StoredProc1->ExecProc();
  Edit1->Text = StoredProc1->ParamByName("PROJ_ID")->AsString;
}
__finally
{
  delete P1;
  delete P2;
}
```

Even if you do not need to add the individual parameter objects at runtime, you may want to access individual parameter objects to assign values to input parameters and to retrieve values from output parameters. You can use the dataset's *ParamByName* method to access individual parameters based on their names. For example, the following code sets the value of an input/output parameter, executes the stored procedure, and retrieves the returned value:

```
with SQLStoredProc1 do
begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger = 103;
SQLDataSet1->ExecSQL();
int Result = SQLDataSet1->ParamByName("IN_OUTVAR")->AsInteger;
```

See Also

Fetching Multiple Result Sets (see page 1597)

Executing Stored Procedures That Don't Return a Result Set (see page 1596)

Preparing Stored Procedures (see page 1607)

3.2.3.6.56 Using Locate

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass *Locate* the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. (Partial-key matching is when the criterion string need only be a prefix of the field value.) For example, the following code moves the cursor to the first row in the *CustTable* where the value in the *Company* column is "Professional Divers, Ltd.":

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
begin
```

```
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.', SearchOptions);
end;
TLocateOptions SearchOptions;
SearchOptions.Clear();
SearchOptions << loPartialKey;
bool LocateSuccess = CustTable->Locate("Company", "Professional Divers, Ltd.",
```

If *Locate* finds a match, the first record containing the match becomes the current record. *Locate* returns *True* if it finds a matching record, *False* if it does not. If a search fails, the current record does not change.

The real power of *Locate* comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are Variants, which means you can specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the Lookup method), or you must construct the Variant array in code using the *VarArrayOf* function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
TLocateOptions Opts;
Opts.Clear();
Opts << loPartialKey;
Variant locvalues[2];
locvalues[0] = Variant("Sight Diver");
locvalues[1] = Variant("P");
CustTable->Locate("Company;Contact", VarArrayOf(locvalues, 1), Opts);
```

Locate uses the fastest possible method to locate matching records. If the columns to search are indexed and the index is compatible with the search options you specify, *Locate* uses the index.

See Also

Using Lookup (see page 1628)

3.2.3.6.57 Using Lookup

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. *Lookup* does not move the cursor to the matching row; it only returns values from it.

In its simplest form, you pass *Lookup* the name of field to search, the field value to match, and the field or fields to return. For example, the following code looks for the first record in the *CustTable* where the value of the *Company* field is "Professional Divers, Ltd.", and returns the company name, a contact person, and a phone number for the company:

```
var
  LookupResults: Variant;
begin
  LookupResults := CustTable.Lookup('Company', 'Professional Divers, Ltd.',
'Company;Contact; Phone');
end;
Variant LookupResults = CustTable->Lookup("Company", "Professional Divers, Ltd",
    "Company;Contact;Phone");
```

Lookup returns values for the specified fields from the first matching record it finds. Values are returned as Variants. If more than one return value is requested, *Lookup* returns a Variant array. If there are no matching records, *Lookup* returns a Null Variant.

The real power of *Lookup* comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual fields in the string items with semicolons.

Because search values are Variants, if you pass multiple values, you must either pass a Variant array as an argument (for example, the return values from the *Lookup* method), or you must construct the Variant array in code using the *VarArrayOf*

function. The following code illustrates a lookup search on multiple columns:

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
'Company; Addr1; Addr2; State; Zip');
end;
Variant LookupResults;
Variant locvalues[2];
Variant v;
locvalues[0] = Variant("Sight Diver");
locvalues[1] = Variant("Kato Paphos");
LookupResults = CustTable->Lookup("Company;City", VarArrayOf(locvalues, 1),
                  "Company;Addr1;Addr2;State;Zip");
// now put the results in a global stringlist (created elsewhere)
pFieldValues->Clear();
for (int i = 0; i < 5; i++) // Lookup call requested 5 fields
{
v = LookupResults.GetElement(i);
if (v.IsNull())
pFieldValues->Add("");
else
pFieldValues->Add(v);
}
```

Like *Locate*, *Lookup* uses the fastest possible method to locate matching records. If the columns to search are indexed, *Lookup* uses the index.

See Also

Using Locate (see page 1627)

3.2.3.6.58 Modifying Data

You can use the following dataset methods to insert, update, and delete data if the read-only *CanModify* property is *True*. *CanModify* is *True* unless the dataset is unidirectional, the database underlying the dataset does not permit read and write privileges, or some other factor intervenes. (Intervening factors include the *ReadOnly* property on some datasets or the *RequestLive* property on *TQuery* components.)

Dataset methods for inserting, updating, and deleting data

Method	Description
Edit	Puts the dataset into *dsEdit* state if it is not already in *dsEdit* or *dsInsert* states.
Append	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in *dsInsert* state.
Insert	Posts any pending data, and puts the dataset in *dsInsert* state.
Post	Attempts to post the new or altered record to the database. If successful, the dataset is put in *dsBrowse* state; if unsuccessful, the dataset remains in its current state.
Cancel	Cancels the current operation and puts the dataset in *dsBrowse* state.
Delete	Deletes the current record and puts the dataset in *dsBrowse* state.

The following topics discuss these methods in greater detail:

- Editing Records (see page 1591)
- Adding New Records (see page 1583)
- Deleting Records (see page 1591)

- Posting Data (see page 1606)
- Canceling Changes (see page 1586)
- Modifying Entire Records (see page 1601)

See Also

Types of Datasets (see page 1635)

Opening and Closing Datasets (see page 1604)

Determining Dataset States (see page 1633)

Navigating Datasets (see page 1634)

Searching Datasets (see page 1630)

Displaying and Editing a Subset of Data Using Filters (see page 1631)

Calculating Fields (see page 1585)

Using TDataSet Descendants (see page 1619)

3.2.3.6.59 Searching Datasets

If a dataset is not unidirectional, you can search against it using the *Locate* and *Lookup* methods. These methods enable you to search on any type of columns in any dataset.

The following topics discuss Locate and Lookup in greater detail:

- Using Locate (see page 1627)
- Using Lookup (see page 1628)

Note: Some *TDataSet* descendants introduce an additional family of methods for searching based on an index. For information about these additional methods, see Using Indexes to Search for Records (see page 1615).

See Also

Modifying Data (see page 1629)

Opening and Closing Datasets (see page 1604)

Determining Dataset States (see page 1633)

Navigating Datasets (see page 1634)

Displaying and Editing a Subset of Data Using Filters (see page 1631)

Types of Datasets (see page 1635)

Calculating Fields (see page 1585)

Using TDataSet Descendants (see page 1619)

3.2.3.6.60 Specifying an Index with IndexName

Use the *IndexName* property to cause an index to be active. Once active, an index determines the order of records in the dataset. (It can also be used as the basis for a master-detail link, an index-based search, or index-based filtering.)

To activate an index, set the *IndexName* property to the name of the index. In some database systems, primary indexes do not have names. To activate one of these indexes, set *IndexName* to a blank string.

At design-time, you can select an index from a list of available indexes by clicking the property's ellipsis button in the **Object**

Inspector. At runtime set *IndexName* using a *String* literal or variable. You can obtain a list of available indexes by calling the *GetIndexNames* method.

The following code sets the index for *CustomersTable* to *CustDescending*:

```
CustomersTable.IndexName := 'CustDescending';
CustomersTable->IndexName = "CustDescending";
```

For information on specifying dBASE non-production index files and dBASE III PLUS-style .NDX files, see Specifying a dBASE index file (see page 1672)

See Also

Obtaining Information About Indexes (see page 1603)

Creating an Index with IndexFieldNames (see page 1581)

3.2.3.6.61 Displaying and Editing a Subset of Data Using Filters

An application is frequently interested in only a subset of records from a dataset. For example, you may be interested in retrieving or viewing only those records for companies based in California in your customer database, or you may want to find a record that contains a particular set of field values. In each case, you can use filters to restrict an application's access to a subset of all records in the dataset.

With unidirectional datasets, you can only limit the records in the dataset by using a query that restricts the records in the dataset. With other *TDataSet* descendants, however, you can define a subset of the data that has already been fetched. To restrict an application's access to a subset of all records in the dataset, you can use filters.

A filter specifies conditions a record must meet to be displayed. Filter conditions can be stipulated in a dataset's Filter property or coded into its OnFilterRecord event handler. Filter conditions are based on the values in any specified number of fields in a dataset, regardless of whether those fields are indexed. For example, to view only those records for companies based in California, a simple filter might require that records contain a value in the State field of "CA".

Note: Filters are applied to every record retrieved in a dataset. When you want to filter large volumes of data, it may be more efficient to use a query to restrict record retrieval, or to set a range on an indexed table rather than using filters.

The following topics describe how to work with filters:

- Enabling and Disabling Filtering (see page 1593)
- Navigating Records in a Filtered Dataset (see page 1603)

See Also

Modifying Data (see page 1629)

Opening and Closing Datasets (see page 1604)

Determining Dataset States (see page 1633)

Navigating Datasets (see page 1634)

Searching Datasets (see page 1630)

Types of Datasets (see page 1635)

Calculating Fields (see page 1585)

Using TDataSet Descendants (see page 1619)

3.2.3.6.62 Understanding Datasets: Overview

The fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. A dataset object represents a set of records from a database organized into a logical table. These records may be the records from a single database table, or they may represent the results of executing a query or stored procedure.

All dataset objects that you use in your database applications descend from TDataSet, and they inherit data fields, properties, events, and methods from this class.

TDataSet is a virtualized dataset, meaning that many of its properties and methods are **virtual** or **abstract**. A *virtual method* is a function or procedure declaration where the implementation of that method can be (and usually is) overridden in descendant objects. An abstract *method* is a function or procedure declaration without an actual implementation. The declaration is a prototype that describes the method (and its parameters and return type, if any) that must be implemented in all descendant dataset objects, but that might be implemented differently by each of them.

Because *TDataSet* contains **abstract** methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of the built-in *TDataSet* descendants and use them in your application, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its abstract methods.

TDataSet defines much that is common to all dataset objects. For example, *TDataSet* defines the basic structure of all datasets: an array of TField components that correspond to actual columns in one or more database tables, lookup fields provided by your application, or calculated fields provided by your application. For information about *TField* components, see "Working with field components." (see page 1877)

The following topics describe how to use the common database functionality introduced by *TDataSet*. Bear in mind, however, that although *TDataSet* introduces the methods for this functionality, not all *TDataSet* dependants implement them. In particular, unidirectional datasets implement only a limited subset.

- Using TDataSet Descendants (see page 1619)
- Determining Dataset States (see page 1633)
- Opening and Closing Datasets (see page 1604)
- Navigating Datasets (see page 1634)
- Searching Datasets (see page 1630)
- Displaying and Editing a Subset of Data Using Filters (see page 1631)
- Modifying Data (see page 1629)
- Calculating Fields (see page 1585)
- Types of Datasets (see page 1635)

See Also

Designing Database Applications (see page 1566)

Using Data Controls (see page 1778)

Working with Field Components (see page 1877)

Creating and Using a Client Dataset (see page 1740)

Connecting to Databases (see page 1506)

Working with ADO Components (see page 1494)

Using Unidirectional Datasets (see page 1823)

Using the Borland Database Engine (see page 1643)

Using Provider Components (see page 1819)

Creating Multi-tiered Applications: Overview (see page 1518)

Using Decision Support Components (see page 1800)

Using XML in Database Applications (see page 1847)

3.2.3.6.63 Determining Dataset States

The *state*—or *mode*—of a dataset determines what can be done to its data. For example, when a dataset is closed, its state is *dsInactive*, meaning that nothing can be done to its data. At runtime, you can examine a dataset's read-only State property to determine its current state. The following table summarizes possible values for the *State* property and what they mean:

Values for the dataset State property

Value	State	Meaning
dsInactive	Inactive	DataSet closed. Its data is unavailable.
dsBrowse	Browse	DataSet open. Its data can be viewed, but not changed. This is the default state of an open dataset.
dsEdit	Edit	DataSet open. The current row can be modified. (not supported on unidirectional datasets)
dsInsert	Insert	DataSet open. A new row is inserted or appended. (not supported on unidirectional datasets)
dsSetKey	SetKey	DataSet open. Enables setting of ranges and key values used for ranges and *GotoKey* operations. (not supported by all datasets)
dsCalcFields	CalcFields	DataSet open. Indicates that an *OnCalcFields* event is under way. Prevents changes to fields that are not calculated.
dsCurValue	CurValue	DataSet open. Indicates that the CurValue property of fields is being fetched for an event handler that responds to errors in applying cached updates.
dsNewValue	NewValue	DataSet open. Indicates that the NewValue property of fields is being fetched for an event handler that responds to errors in applying cached updates.
dsOldValue	OldValue	DataSet open. Indicates that the OldValue property of fields is being fetched for an event handler that responds to errors in applying cached updates.
dsFilter	Filter	DataSet open. Indicates that a filter operation is under way. A restricted set of data can be viewed, and no data can be changed. (not supported on unidirectional datasets)
dsBlockRead	Block Read	DataSet open. Data-aware controls are not updated and events are not triggered when the current record changes.
dsInternalCalc	Internal Calc	DataSet open. An *OnCalcFields* event is underway for calculated values that are stored with the record. (client datasets only)
dsOpening	Opening	DataSet is in the process of opening but has not finished. This state occurs when the dataset is opened for asynchronous fetching.

Typically, an application checks the dataset state to determine when to perform certain tasks. For example, you might check for the *dsEdit* or *dsInsert* state to ascertain whether you need to post updates.

Note: Whenever a dataset's state changes, the *OnStateChange* event is called for any data source components associated with the dataset. For more information about data source components and *OnStateChange*, see Responding to Changes Mediated by the Data Source (see page 1776).

See Also

Opening and Closing Datasets (see page 1604)

Navigating Datasets (see page 1634)

Searching Datasets (see page 1630)

Displaying and Editing a Subset of Data Using Filters (see page 1631)

Modifying Data (see page 1629)

Types of Datasets (see page 1635)

Calculating Fields (see page 1585)

Using TDataSet Descendants (see page 1619)

3.2.3.6.64 Navigating Datasets

Each active dataset has a *cursor*, or pointer, to the current row in the dataset. The *current row* in a dataset is the one whose field values currently show in single-field, data-aware controls on a form, such as TDBEdit, *TDBLabel*, and TDBMemo. If the dataset supports editing, the current record contains the values that can be manipulated by edit, insert, and delete methods.

You can change the current row by moving the cursor to point at a different row. The following table lists methods you can use in application code to move to different records:

Navigational methods of datasets

Method	Moves the Cursor to
First	The first row in a dataset.
Last	The last row in a dataset. (not available for unidirectional datasets)
Next	The next row in a dataset.
Prior	The previous row in a dataset. (not available for unidirectional datasets)
MoveBy	A specified number of rows forward or back in a dataset.

The data-aware, visual component TDBNavigator encapsulates these methods as buttons that users can click to move among records at runtime. For information about the navigator component, see Navigating and manipulating records (see page 1767).

Whenever you change the current record using one of these methods (or by other methods that navigate based on a search criterion), the dataset receives two events: BeforeScroll (before leaving the current record) and AfterScroll (after arriving at the new record). You can use these events to update your user interface (for example, to update a status bar that indicates information about the current record).

TDataSet also defines two boolean properties that provide useful information when iterating through the records in a dataset.

Navigational properties of datasets

Property	Description
BOF (Beginning-of-file)	*True*: the cursor is at the first row in the dataset. *False*: the cursor is not known to be at the first row in the dataset
EOF (End-of-file)	*True*: the cursor is at the last row in the dataset. *False*: the cursor is not known to be at the first row in the dataset

The following topics discuss these properties and methods in more detail:

- Using the First and Last methods (see page 1623)
- Using the Next and Prior methods (see page 1624)
- Using the MoveBy method (see page 1623)
- Using the Eof and Bof Properties (see page 1621)

- Marking and Returning to Records (see page 1599)

See Also

Modifying Data (see page 1629)

Types of Datasets (see page 1635)

Opening and Closing Datasets (see page 1604)

Determining Dataset States (see page 1633)

Searching Datasets (see page 1630)

Displaying and Editing a Subset of Data Using Filters (see page 1631)

Navigating Records in a Filtered Dataset (see page 1603)

Calculating Fields (see page 1585)

Using TDataSet Descendants (see page 1619)

3.2.3.6.65 Writing an OnFilterRecord Event Handler

You can write code to filter records using the OnFilterRecord events generated by the dataset for each record it retrieves. This event handler implements a test that determines if a record should be included in those that are visible to the application.

To indicate whether a record passes the filter condition, your *OnFilterRecord* handler sets its *Accept* parameter to *True* to include a record, or *False* to exclude it. For example, the following filter displays only those records with the State field set to "CA":

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['State'].AsString = 'CA';
end;
void __fastcall TForm1::Table1FilterRecord(TDataSet *DataSet; bool &Accept)
{
  Accept = DataSet->FieldByName["State"]->AsString == "CA";
}
```

When filtering is enabled, an *OnFilterRecord* event is generated for each record retrieved. The event handler tests each record, and only those that meet the filter's conditions are displayed. Because the *OnFilterRecord* event is generated for every record in a dataset, you should keep the event handler as tightly coded as possible to avoid adversely affecting the performance.

You can code any number of *OnFilterRecord* event handlers and switch among them at runtime. For example, the following statements switch to an *OnFilterRecord* event handler called *NewYorkFilter*:

```
DataSet1.OnFilterRecord := NewYorkFilter;
Refresh;
DataSet1->OnFilterRecord = NewYorkFilter;
()
```

See Also

Setting the Filter Property (see page 1608)

3.2.3.6.66 Types of Datasets

Using TDataSet descendants (see page 1619) classifies *TDataSet* descendants by the method they use to access their data. Another useful way to classify *TDataSet* descendants is to consider the type of server data they represent. Viewed this way, there are three basic classes of datasets:

Table type datasets (see page 1620): Table type datasets represent a single table from the database server, including all of its rows and columns. Table type datasets include TTable, TADOTable, TSQLTable, and TIBTable.

Table type datasets let you take advantage of indexes defined on the server. Because there is a one-to-one correspondence between database table and dataset, you can use server indexes that are defined for the database table. Indexes allow your application to sort the records in the table, speed searches and lookups, and can form the basis of a master/detail relationship. Some table type datasets also take advantage of the one-to-one relationship between dataset and database table to let you perform table-level operations such as creating and deleting database tables.

Query-type datasets (see page 1618): Query-type datasets represent a single SQL command, or query. Queries can represent the result set from executing a command (typically a SELECT statement), or they can execute a command that does not return any records (for example, an UPDATE statement). Query-type datasets include TQuery, TADOQuery, TSQLQuery, and TIBQuery.

To use a query-type dataset effectively, you must be familiar with SQL and your server's SQL implementation, including limitations and extensions to the SQL-92 standard. If you are new to SQL, you may want to purchase a third party book that covers SQL in-depth. One of the best is *Understanding the New SQL: A Complete Guide*, by Jim Melton and Alan R. Simpson, Morgan Kaufmann Publishers.

Stored procedure-type datasets (see page 1618): Stored procedure-type datasets represent a stored procedure on the database server. Stored procedure-type datasets include TStoredProc, TADOStoredProc, TSQLStoredProc, and TIBStoredProc.

A stored procedure is a self-contained program written in the procedure and trigger language specific to the database system used. They typically handle frequently repeated database-related tasks, and are especially useful for operations that act on large numbers of records or that use aggregate or mathematical functions. Using stored procedures typically improves the performance of a database application by:

- Taking advantage of the server's usually greater processing power and speed.
- Reducing network traffic by moving processing to the server.

Stored procedures may or may not return data. Those that return data may return it as a cursor (similar to the results of a SELECT query), as multiple cursors (effectively returning multiple datasets), or they may return data in output parameters. These differences depend in part on the server: Some servers do not allow stored procedures to return data, or only allow output parameters. Some servers do not support stored procedures at all. See your server documentation to determine what is available.

Note: You can usually use a query-type dataset to execute stored procedures because most servers provide extensions to SQL for working with stored procedures. Each server, however, uses its own syntax for this. If you choose to use a query-type dataset instead of a stored procedure-type dataset, see your server documentation for the necessary syntax.

In addition to the datasets that fall neatly into these three categories, *TDataSet* has some descendants that fit into more than one category:

- TADODataSet and TSQLDataSet*have a CommandType property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are most similar to query-type datasets, although TADODataSet lets you specify an index like a table type dataset.*
- TClientDataSet represents the data from another dataset. As such, it can represent a table, query, or stored procedure. *TClientDataSet* behaves most like a table type dataset, because of its index support. However, it also has some of the features of queries and stored procedures: the management of parameters and the ability to execute without retrieving a result set.
- Some other client datasets (like TBDEClientDataSet) have a *CommandType* property that lets you specify whether they represent a table, query, or stored procedure. Property and method names are like *TClientDataSet*, including parameter support, indexes, and the ability to execute without retrieving a result set.
- TIBDataSet can represent both queries and stored procedures. In fact, it can represent multiple queries and stored procedures simultaneously, with separate properties for each.

See Also

Opening and Closing Datasets (see page 1604)

Determining Dataset States (see page 1633)

Navigating Datasets (see page 1634)

Searching Datasets (see page 1630)

Displaying and Editing a Subset of Data Using Filters (see page 1631)

Modifying Data (see page 1629)

Database Architecture (see page 1562)

Calculating Fields (see page 1585)

3.2.3.7 Using the Borland Database Engine

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Topics

Name	Description
BDE-based Architecture (see page 1638)	**Warning:** The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress. When using the BDE, your application uses a variation of the general database architecture described in Database Architecture (see page 1562). In addition to the user interface elements, datasource, and datasets common to all Delphi database applications, A BDE-based application can include • One or more database components to control transactions and to manage database connections. • One or more session components... more (see page 1638)
Executing an Update Statement (see page 1639)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress. The *ExecSQL* method for an update component manually applies updates for the current record. Unlike the *Apply* method, *ExecSQL* does not bind parameters in the SQL statement before executing it. The *ExecSQL* method is most often called from within a handler for the *OnUpdateRecord* event (when using the BDE) or the *BeforeUpdateRecord* event (when using a client dataset). Because... more (see page 1639)
Using Multiple Update Objects (see page 1641)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress. When more than one base table referenced in the update dataset needs to be updated, you need to use multiple update objects: one for each base table updated. Because the dataset component's *UpdateObject* only allows one update object to be associated with the dataset, you must associate each update object with a dataset by setting its DataSet property to... more (see page 1641)
Using the Borland Database Engine (see page 1643)	The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers to connect to different databases. Depending on your edition of Delphi, you can use the drivers for local databases (Paradox, dBASE, FoxPro, and Access) and an ODBC adapter that lets you supply your own ODBC drivers. When deploying BDE-based applications, you must... more (see page 1643)
Connecting to databases with TDatabase (see page 1644)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Managing database sessions (see page 1649)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.
Using BDE-enabled datasets (see page 1666)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.
Using TBatchMove (see page 1678)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.
Using the BDE to cache updates (see page 1685)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.
Using transactions with the BDE (see page 1698)	The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

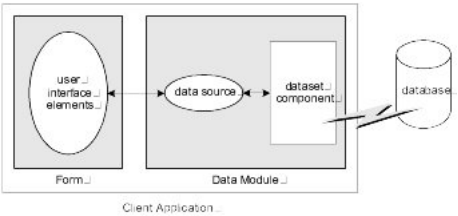
3.2.3.7.1 BDE-based Architecture

Warning: The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

When using the BDE, your application uses a variation of the general database architecture described in Database Architecture (see page 1562). In addition to the user interface elements, datasource, and datasets common to all Delphi database applications, A BDE-based application can include

- One or more database components to control transactions and to manage database connections.
- One or more session components to isolate data access operations such as database connections, and to manage groups of databases.

The relationships between the components in a BDE-based application are illustrated in the following figure:



The following topics provide additional information about these components:

- Using BDE-enabled Datasets (see page 1675)
- Connecting to Databases with TDatabase (see page 1646)
- Managing Database Sessions (see page 1664)

See Also

Using TBatchMove (see page 1683)

Tools for Working with the BDE (see page 1681)

Using Transactions with the BDE (see page 1699)

The Data Dictionary (see page 1680)

3.2.3.7.2 Executing an Update Statement

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

The *ExecSQL* method for an update component manually applies updates for the current record. Unlike the *Apply* method, *ExecSQL* does not bind parameters in the SQL statement before executing it. The *ExecSQL* method is most often called from within a handler for the *OnUpdateRecord* event (when using the BDE) or the *BeforeUpdateRecord* event (when using a client dataset).

Because *ExecSQL* does not bind parameter values, it is used primarily when the update object's SQL statements do not include parameters. You can use *Apply* instead, even when there are no parameters, but *ExecSQL* is more efficient because it does not check for parameters.

If the SQL statements include parameters, you can still call *ExecSQL,* but only after explicitly binding parameters. If you are using the BDE to cache updates, you can explicitly bind parameters by setting the update object's *DataSet* property and then calling its *SetParams* method. When using a client dataset to cache updates, you must supply parameters to the underlying query object maintained by *TUpdateSQL*. For information on how to do this, see Using an update component's Query property (see page 1640).

Warning: If you use the dataset's *UpdateObject* property to associate dataset and update object, *ExecSQL* is called automatically. In that case, do not call *ExecSQL* in an *OnUpdateRecord* or *BeforeUpdateRecord* event handler as this will result in a second attempt to apply the current record's update.

OnUpdateRecord and *BeforeUpdateRecord* event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *ExecSQL* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  with UpdateSQL1 do
  begin
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ExecSQL(UpdateKind);
    Applied := True;
  end;
end;
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  UpdateSQL1->DatabaseName = pSrcDS->DatabaseName;
  UpdateSQL1->SessionName = pSrcDS->SessionName;
  UpdateSQL1->ExecSQL(UpdateKind);
  Applied = true;
}
```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Topics

Name	Description
Using an Update Component's Query Property (see page 1640)	The *Query* property of an update component provides access to the query components that implement its *DeleteSQL*, *InsertSQL*, and *ModifySQL* statements. In most applications, there is no need to access these query components directly: you can use the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties to specify the statements these queries execute, and execute them by calling the update object's *Apply* or *ExecSQL* method. There are times, however, when you may need to directly manipulate the query component. In particular, the *Query* property is useful when you want to supply your own values for parameters in the SQL... more (see page 1640)

See Also

Calling the Apply Method (see page 1642)

3.2.3.7.2.1 Using an Update Component's Query Property

The *Query* property of an update component provides access to the query components that implement its *DeleteSQL*, *InsertSQL*, and *ModifySQL* statements. In most applications, there is no need to access these query components directly: you can use the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties to specify the statements these queries execute, and execute them by calling the update object's *Apply* or *ExecSQL* method. There are times, however, when you may need to directly manipulate the query component. In particular, the *Query* property is useful when you want to supply your own values for parameters in the SQL statements rather than relying on the update object's automatic parameter binding to old and new field values.

Note: The *Query* property is only accessible at runtime.

The *Query* property is indexed on a *TUpdateKind* value:

- Using an index of *ukModify* accesses the query that updates existing records.
- Using an index of *ukInsert* accesses the query that inserts new records.
- Using an index of *ukDelete* accesses the query that deletes records.

The following shows how to use the *Query* property to supply parameter values that can't be bound automatically:

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  UpdateSQL1.DataSet := DeltaDS; { required for the automatic parameter substitution }
  with UpdateSQL1.Query[UpdateKind] do
  begin
    { Make sure the query has the correct DatabaseName and SessionName }
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    ParamByName('TimeOfUpdate').Value = Now;
  end;
  UpdateSQL1.Apply(UpdateKind); { now perform automatic substitutions and execute }
  Applied := True;
end;
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  UpdateSQL1->DataSet = DeltaDS; // required for the automatic parameter substitution
  TQuery *pQuery = UpdateSQL1->Query[UpdateKind]; // access the query
  // make sure the query has the correct DatabaseName and SessionName
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  pQuery->DatabaseName = pSrcDS->DatabaseName;
  pQuery->SessionName = pSrcDS->SessionName;
  // now substitute values for custom parameters
  pQuery->ParamByName("TimeOfLastUpdate")->Value = Now();
  UpdateSQL1->Apply(UpdateKind); // now do automatic substitution and execute
  Applied = true;
```

```
}
```

See Also

Calling the Apply Method (see page 1642)

3.2.3.7.3 Using Multiple Update Objects

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

When more than one base table referenced in the update dataset needs to be updated, you need to use multiple update objects: one for each base table updated. Because the dataset component's *UpdateObject* only allows one update object to be associated with the dataset, you must associate each update object with a dataset by setting its DataSet property to the name of the dataset.

Tip: When using multiple update objects, you can use *TBDEClientDataSet* instead of *TClientDataSet* with an external provider. This is because you do not need to set the source dataset's *UpdateObject* property.

The *DataSet* property for update objects is not available at design time in the **Object Inspector**. You can only set this property at runtime.

```
UpdateSQL1.DataSet := Query1;
UpdateSQL1->DataSet = Query1;
```

The update object uses this dataset to obtain original and updated field values for parameter substitution and, if it is a BDE-enabled dataset, to identify the session and database to use when applying the updates. So that parameter substitution will work correctly, the update object's *DataSet* property must be the dataset that contains the updated field values. When using the BDE-enabled dataset to cache updates, this is the BDE-enabled dataset itself. When using a client dataset, this is a client dataset that is provided as a parameter to the *BeforeUpdateRecord* event handler.

When the update object has not been assigned to the dataset's *UpdateObject* property, its SQL statements are not automatically executed when you call *ApplyUpdates*. To update records, you must manually call the update object from an OnUpdateRecord event handler (when using the BDE to cache updates) or a BeforeUpdateRecord event handler (when using a client dataset). In the event handler, the minimum actions you need to take are

- If you are using a client dataset to cache updates, you must be sure that the updates object's DatabaseName and SessionName properties are set to the DatabaseName and SessionName properties of the source dataset.
- The event handler must call the update object's ExecSQL or Apply method. This invokes the update object for each record that requires updating. For more information about executing update statements, see Executing the SQL statements (see page 1643).
- Set the event handler's *UpdateAction* parameter to *uaApplied* (*OnUpdateRecord*) or the *Applied* parameter to *True* (*BeforeUpdateRecord*).

You may optionally perform data validation, data modification, or other operations that depend on each record's update.

Warning: If you call an update object's *ExecSQL* or *Apply* method in an *OnUpdateRecord* event handler, be sure that you do not set the dataset's *UpdateObject* property to that update object. Otherwise, this will result in a second attempt to apply each record's update.

Topics

Name	Description
Calling the Apply Method (see page 1642)	The *Apply* method for an update component manually applies updates for the current record. There are two steps involved in this process: 1. Initial and edited field values for the record are bound to parameters in the appropriate SQL statement. 2. The SQL statement is executed. Call the *Apply* method to apply the update for the current record in the update cache. The *Apply* method is most often called from within a handler for the dataset's *OnUpdateRecord* event or from a provider's *BeforeUpdateRecord* event handler. **Warning:** If you use the dataset's *UpdateObject* property to associate dataset and update object, *Apply* is called... more (see page 1642)
Executing the SQL Statements (see page 1643)	When you use multiple update objects, you do not associate the update objects with a dataset by setting its *UpdateObject* property. As a result, the appropriate statements are not automatically executed when you apply updates. Instead, you must explicitly invoke the update object in code. There are two ways to invoke the update object. Which way you choose depends on whether the SQL statement uses parameters to represent field values (see page 1689): • If the SQL statement to execute uses parameters, call the Apply method (see page 1642). • If the SQL statement to execute does not use parameters, it is more efficient to call... more (see page 1643)

See Also

Creating SQL Statements for Update Components (see page 1696)

3.2.3.7.3.1 Calling the Apply Method

The *Apply* method for an update component manually applies updates for the current record. There are two steps involved in this process:

1. Initial and edited field values for the record are bound to parameters in the appropriate SQL statement.
2. The SQL statement is executed.

Call the *Apply* method to apply the update for the current record in the update cache. The *Apply* method is most often called from within a handler for the dataset's *OnUpdateRecord* event or from a provider's *BeforeUpdateRecord* event handler.

Warning: If you use the dataset's *UpdateObject* property to associate dataset and update object, *Apply* is called automatically. In that case, do not call *Apply* in an *OnUpdateRecord* event handler as this will result in a second attempt to apply the current record's update.

OnUpdateRecord event handlers indicate the type of update that needs to be applied with an *UpdateKind* parameter of type *TUpdateKind*. You must pass this parameter to the *Apply* method to indicate which update SQL statement to use. The following code illustrates this using a *BeforeUpdateRecord* event handler:

```
procedure TForm1.BDEClientDataSet1BeforeUpdateRecord(Sender: TObject; SourceDS: TDataSet;
          DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  with UpdateSQL1 do
  begin
    DataSet := DeltaDS;
    DatabaseName := (SourceDS as TDBDataSet).DatabaseName;
    SessionName := (SourceDS as TDBDataSet).SessionName;
    Apply(UpdateKind);
```

```
    Applied := True;
  end;
end;
void __fastcall TForm1::BDEClientDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  UpdateSQL1->DataSet = DeltaDS;
  TDBDataSet *pSrcDS = dynamic_cast<TDBDataSet *>(SourceDS);
  UpdateSQL1->DatabaseName = pSrcDS->DatabaseName;
  UpdateSQL1->SessionName = pSrcDS->SessionName;
  UpdateSQL1->Apply(UpdateKind);
  Applied = true;
}
```

See Also

Executing an Update Statement (see page 1639)

3.2.3.7.3.2 Executing the SQL Statements

When you use multiple update objects, you do not associate the update objects with a dataset by setting its *UpdateObject* property. As a result, the appropriate statements are not automatically executed when you apply updates. Instead, you must explicitly invoke the update object in code.

There are two ways to invoke the update object. Which way you choose depends on whether the SQL statement uses parameters to represent field values (see page 1689):

- If the SQL statement to execute uses parameters, call the Apply method (see page 1642).
- If the SQL statement to execute does not use parameters, it is more efficient to call the ExecSQL method.

  Note: If the SQL statement uses parameters other than the built-in types (for the original and updated field values), you must manually supply parameter values instead of relying on the parameter substitution provided by the *Apply* method. See Using an update component's Query property (see page 1640) for information on manually providing parameter values.

See Also

Creating SQL Statements for Update Components (see page 1696)

3.2.3.7.4 Using the Borland Database Engine

The Borland Database Engine (BDE) is a data-access mechanism that can be shared by several applications. The BDE defines a powerful library of API calls that can create, restructure, fetch data from, update, and otherwise manipulate local and remote database servers. The BDE provides a uniform interface to access a wide variety of database servers, using drivers to connect to different databases. Depending on your edition of Delphi, you can use the drivers for local databases (Paradox, dBASE, FoxPro, and Access) and an ODBC adapter that lets you supply your own ODBC drivers.

When deploying BDE-based applications, you must include the BDE with your application. While this increases the size of the application and the complexity of deployment, the BDE can be shared with other BDE-based applications and provides a broad range of support for database manipulation. Although you can use the BDE's API directly in your application, the components on the BDE category of the **Tool palette** wrap most of this functionality for you.

See Also

Understanding Datasets (see page 1632)

Using Data Controls (see page 1778)

Working with Field Components (see page 1877)

Connecting to Databases (see page 1506)

Designing Database Applications (see page 1566)

Creating and Using a Client Dataset (see page 1740)

Working With ADO Components (see page 1494)

Using Unidirectional Datasets (see page 1823)

Using Provider Components (see page 1819)

Creating Multi-tiered Applications: Overview (see page 1518)

Using Decision Support Components (see page 1800)

Using XML in Database Applications (see page 1847)

3.2.3.7.5 Connecting to databases with TDatabase

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Topics

Name	Description
Associating a Database Component with a Session (see page 1645)	All database components must be associated with a BDE session. Use the SessionName, establish this association. When you first create a database component at design time, *SessionName* is set to "Default", meaning that it is associated with the default session component that is referenced by the global Session variable. Multi-threaded or reentrant BDE applications may require more than one session. If you need to use multiple sessions, add TSession components for each session. Then, associate your dataset with a session component by setting the SessionName property to a session component's SessionName property. At runtime, you can access the session component... more (see page 1645)
Using TDatabase to Connect to Databases (see page 1646)	When a Delphi application uses the Borland Database Engine (BDE) to connect to a database, that connection is encapsulated by a TDatabase component. A database component represents the connection to a single database in the context of a BDE session. *TDatabase* performs many of the same tasks as and shares many common properties, methods, and events with other database connection components. These commonalities are described in Connecting to databases (see page 1506). In addition to the common properties, methods, and events, *TDatabase* introduces many BDE-specific features. These features are described in the following topics: • Associating a Database Component with a Session (see page 1645)... more (see page 1646)
Opening a Connection Using TDataBase (see page 1646)	As with all database connection components, to connect to a database using *TDatabase*, you set the *Connected* property to *True* or call the *Open* method. This process is described in Connecting to a database server (see page 1509). Once a database connection is established the connection is maintained as long as there is at least one active dataset. When there are no more active datasets, the connection is dropped unless the database component's KeepConnection property is *True*. When you connect to a remote database server from an application, the application uses the BDE and the CodeGear SQL Links driver to... more (see page 1646)
Understanding Database and Session Component Interactions (see page 1647)	In general, session component properties provide global, default behaviors that apply to all implicit database components created at runtime. For example, the controlling session's KeepConnections property determines whether a database connection is maintained even if its associated datasets are closed (the default), or if the connections are dropped when all its datasets are closed. Similarly, the default *OnPasswordevent for a session guarantees that when an application attempts to attach to a database on a server that requires a password, it displays a standard password prompt dialog box.* *Session methods apply somewhat differently. TSession methods affect all database components, regardless... more (see page 1647)*

Using Database Components in Data Modules (see page 1647)	You can safely place database components in data modules. If you put a data module that contains a database component into the Object Repository, however, and you want other users to be able to inherit from it, you must set the HandleShared property of the database component to *True* to prevent global name space conflicts.
Setting BDE Alias Parameters (see page 1648)	At design time you can create or edit connection parameters in three ways: • Use the Database Explorer to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files. • Double-click the *Params* property in the **Object Inspector** to invoke the String List editor (see page 2140). • Double-click a database component in a data module or form to invoke the Database Properties editor. All of these methods edit the Params property for the database component. *Params* is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical... more (see page 1648)
Identifying the Database (see page 1648)	*AliasName* and *DriverName* are mutually exclusive properties that identify the database server to which the *TDatabase* component connects. AliasName specifies the name of an existing BDE alias to use for the database component. The alias appears in subsequent drop-down lists for dataset components so that you can link them to a particular database component. If you specify *AliasName* for a database component, any value already assigned to *DriverName* is cleared because a driver name is always part of a BDE alias. You create and edit BDE aliases using the Database Explorer . For more information about creating and maintaining BDE... more (see page 1648)

3.2.3.7.5.1 Associating a Database Component with a Session

All database components must be associated with a BDE session. Use the SessionName, establish this association. When you first create a database component at design time, *SessionName* is set to "Default", meaning that it is associated with the default session component that is referenced by the global Session variable.

Multi-threaded or reentrant BDE applications may require more than one session. If you need to use multiple sessions, add TSession components for each session. Then, associate your dataset with a session component by setting the SessionName property to a session component's SessionName property.

At runtime, you can access the session component with which the database is associated by reading the Session property. If *SessionName* is blank or "Default", then the *Session* property references the same *TSession* instance referenced by the global *Session* variable. *Session* enables applications to access the properties, methods, and events of a database component's parent session component without knowing the session's actual name.

For more information about BDE sessions, see Managing database sessions (see page 1664).

If you are using an implicit database component (see page 1505), the session for that database component is the one specified by the dataset's SessionName property.

See Also

Understanding Database and Session Component Interactions (see page 1647)

Identifying the Database (see page 1648)

Opening a Connection Using TDataBase (see page 1646)

Using Database Components in Data Modules (see page 1647)

Applying Cached Updates with a Database Component Method (see page 1695)

3.2.3.7.5.2 Using TDatabase to Connect to Databases

When a Delphi application uses the Borland Database Engine (BDE) to connect to a database, that connection is encapsulated by a TDatabase component. A database component represents the connection to a single database in the context of a BDE session.

TDatabase performs many of the same tasks as and shares many common properties, methods, and events with other database connection components. These commonalities are described in Connecting to databases (see page 1506).

In addition to the common properties, methods, and events, *TDatabase* introduces many BDE-specific features. These features are described in the following topics:

- Associating a Database Component with a Session (see page 1645)
- Understanding Database and Session Component Interactions (see page 1647)
- Identifying the Database (see page 1648)
- Opening a Connection Using TDatabase (see page 1646)
- Using Database Components in Data Modules (see page 1647)
- Applying Cached Updates Using a Database (see page 1687).

See Also

Using BDE Enabled Datasets (see page 1675)

Managing Database Sessions Using TSession (see page 1664)

3.2.3.7.5.3 Opening a Connection Using TDataBase

As with all database connection components, to connect to a database using *TDatabase*, you set the *Connected* property to *True* or call the *Open* method. This process is described in Connecting to a database server (see page 1509). Once a database connection is established the connection is maintained as long as there is at least one active dataset. When there are no more active datasets, the connection is dropped unless the database component's KeepConnection property is *True*.

When you connect to a remote database server from an application, the application uses the BDE and the CodeGear SQL Links driver to establish the connection. (The BDE can also communicate with an ODBC driver that you supply.) You need to configure the SQL Links or ODBC driver for your application prior to making the connection. SQL Links and ODBC parameters are stored in the Params property of a database component. For information about SQL Links parameters, see the online *SQL Links* User's Guide. To edit the *Params* property, see Setting BDE alias parameters (see page 1648)

Working with network protocols

As part of configuring the appropriate SQL Links or ODBC driver, you may need to specify the network protocol used by the server, such as SPX/IPX or TCP/IP, depending on the driver's configuration options. In most cases, network protocol configuration is handled using a server's client setup software. For ODBC it may also be necessary to check the driver setup using the ODBC driver manager.

Establishing an initial connection between client and server can be problematic. The following troubleshooting checklist should be helpful if you encounter difficulties:

- Is your server's client-side connection properly configured?
- Are the DLLs for your connection and database drivers in the search path?
- If you are using TCP/IP:
- Is your TCP/IP communications software installed? Is the proper WINSOCK.DLL installed?
- Is the server's IP address registered in the client's HOSTS file?
- Is the Domain Name Services (DNS) properly configured?

- Can you ping the server?

For more troubleshooting information, see the online *SQL Links User's Guide* and your server documentation.

Using ODBC

An application can use ODBC data sources (for example, Btrieve). An ODBC driver connection requires

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.

See Also

Associating a Database Component with a Session (see page 1645)

Understanding Database and Session Component Interactions (see page 1647)

Identifying the Database (see page 1648)

Using Database Components in Data Modules (see page 1647)

Applying Cached Updates with a Database Component Method (see page 1695)

3.2.3.7.5.4 Understanding Database and Session Component Interactions

In general, session component properties provide global, default behaviors that apply to all implicit database components created at runtime. For example, the controlling session's KeepConnections property determines whether a database connection is maintained even if its associated datasets are closed (the default), or if the connections are dropped when all its datasets are closed. Similarly, the default *OnPasswordevent for a session guarantees that when an application attempts to attach to a database on a server that requires a password, it displays a standard password prompt dialog box.*

Session methods apply somewhat differently. TSession methods affect all database components, regardless of whether they are explicitly created or instantiated implicitly by a dataset. For example, the session method DropConnectionscloses all datasets belonging to a session's database components, and then drops all database connections, even if the KeepConnection property for individual database components is True.

Database component methods apply only to the datasets associated with a given database component. For example, suppose the database component Database1 is associated with the default session. Database1.CloseDataSets() closes only those datasets associated with Database1. Open datasets belonging to other database components within the default session remain open.

See Also

Associating a Database Component with a Session (see page 1645)

Identifying the Database (see page 1648)

Opening a Connection Using TDataBase (see page 1646)

Using Database Components in Data Modules (see page 1647)

Applying Cached Updates with a Database Component Method (see page 1695)

3.2.3.7.5.5 Using Database Components in Data Modules

You can safely place database components in data modules. If you put a data module that contains a database component into the Object Repository, however, and you want other users to be able to inherit from it, you must set the HandleShared property of the database component to *True* to prevent global name space conflicts.

See Also

Associating a Database Component with a Session (see page 1645)

Understanding Database and Session Component Interactions (see page 1647)

Identifying the Database (see page 1648)

Opening a Connection Using TDataBase (see page 1646)

Applying Cached Updates with a Database Component Method (see page 1695)

3.2.3.7.5.6 Setting BDE Alias Parameters

At design time you can create or edit connection parameters in three ways:

- Use the Database Explorer to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files.
- Double-click the *Params* property in the **Object Inspector** to invoke the String List editor (see page 2140).
- Double-click a database component in a data module or form to invoke the Database Properties editor.

All of these methods edit the Params property for the database component. *Params* is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical connection parameters include path statement, server name, schema caching size, language driver, and SQL query mode.

When you first invoke the Database Properties editor, the parameters for the BDE alias are not visible. To see the current settings, click Defaults. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click Clear. Changes you make take effect only when you click OK.

At runtime, an application can set alias parameters only by editing the *Params* property directly.

3.2.3.7.5.7 Identifying the Database

AliasName and *DriverName* are mutually exclusive properties that identify the database server to which the *TDatabase* component connects.

AliasName specifies the name of an existing BDE alias to use for the database component. The alias appears in subsequent drop-down lists for dataset components so that you can link them to a particular database component. If you specify *AliasName* for a database component, any value already assigned to *DriverName* is cleared because a driver name is always part of a BDE alias.

You create and edit BDE aliases using the Database Explorer . For more information about creating and maintaining BDE aliases, see the online documentation for these utilities.

DriverName*is the name of a BDE driver. A driver name is one parameter in a BDE alias, but you may specify a driver name instead of an alias when you create a local BDE alias for a database component using the DatabaseName property. If you specify DriverName, any value already assigned to AliasName is cleared to avoid potential conflicts between the driver name you specify and the driver name that is part of the BDE alias identified in AliasName.*

DatabaseName lets you provide your own name for a database connection. The name you supply is in addition to AliasName or DriverName, and is local to your application. DatabaseName can be a BDE alias, or, for Paradox and dBASE files, a fully-qualified path name. Like AliasName, DatabaseName appears in subsequent drop-down lists for dataset components to let you link them to database components.

At design time, to specify a BDE alias, assign a BDE driver, or create a local BDE alias, double-click a database component to invoke the Database Properties editor.

You can enter a DatabaseName in the Name edit box in the properties editor. You can enter an existing BDE alias name in the Alias name combo box for the Alias property, or you can choose from existing aliases in the drop-down list. The Driver name combo box enables you to enter the name of an existing BDE driver for the DriverName property, or you can choose from existing driver names in the drop-down list.

Note: *The Database Properties editor also lets you view and set BDE connection parameters, and set the states of the*

LoginPrompt and KeepConnection properties. For information on connection parameters, see Setting BDE Alias Parameters (see page 1648). For information on LoginPrompt, see Controlling Server Login (see page 1507). For information on KeepConnection see Opening a Connection Using TDatabase (see page 1646).

See Also

Associating a Database Component with a Session (see page 1645)

Understanding Database and Session Component Interactions (see page 1647)

Opening a Connection Using TDataBase (see page 1646)

Using Database Components in Data Modules (see page 1647)

Applying Cached Updates with a Database Component Method (see page 1695)

3.2.3.7.6 Managing database sessions

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Topics

Name	Description
Activating a Session (see page 1651)	Active is a Boolean property that determines if database and dataset components associated with a session are open. You can use this property to read the current state of a session's database and dataset connections, or to change it. If *Active* is *False* (the default), all databases and datasets associated with the session are closed. If *True*, databases and datasets are open. A session is activated when it is first created, and subsequently, whenever its *Active* property is changed to *True* from *False* (for example, when a database or dataset is associated with a session is opened and there... more (see page 1651)
Creating Additional Sessions (see page 1652)	You can create sessions to supplement the default session. At design time, you can place additional sessions on a data module (or form), set their properties in the **Object Inspector**, write event handlers for them, and write code that calls their methods. You can also create sessions, set their properties, and call their methods at runtime. **Note:** Creating additional sessions is optional unless an application runs concurrent queries against a database or the application is multi-threaded.
Iterating Through a Session's Database Components (see page 1652)	You can use two session component properties, *Databases* and *DatabaseCount*, to cycle through all the active database components associated with a session. *Databases* is an array of all currently active database components associated with a session. *DatabaseCount* is the number of databases in that array. As connections are opened or closed during a session's life-span, the values of *Databases* and *DatabaseCount* change. For example, if a session's *KeepConnections* property is *False* and all database components are created as needed at runtime, each time a unique database is opened, *DatabaseCount* increases by one. Each time a unique database is closed,... more (see page 1652)
Opening Database Connections (see page 1653)	To open a database connection within a session, call the OpenDatabase method. *OpenDatabase* takes one parameter, the name of the database to open. This name is a BDE alias or the name of a database component. For Paradox or dBASE, the name can also be a fully qualified path name. For example, the following statement uses the default session and attempts to open a database connection for the database pointed to by the DBDEMOS alias:
Searching for a Database Connection (see page 1654)	Use a session's FindDatabase method to determine whether a specified database component is already associated with a session. *FindDatabase* takes one parameter, the name of the database to search for. This name is a BDE alias or database component name. For Paradox or dBASE, it can also be a fully-qualified path name. *FindDatabase* returns the database component if it finds a match. Otherwise it returns **nil**. The following code searches the default session for a database component using the DBDEMOS alias, and if it is not found, creates one and opens it:

Specifying Paradox Directory Locations (see page 1654)	Two session component properties, *NetFileDir* and *PrivateDir*, are specific to applications that work with Paradox tables. *NetFileDir* specifies the directory that contains the Paradox network control file, PDOXUSRS.NET. This file governs sharing of Paradox tables on network drives. All applications that need to share Paradox tables must specify the same directory for the network control file (typically a directory on a network file server). Delphi derives a value for *NetFileDir* from the Borland Database Engine (BDE) configuration file for a given database alias. If you set *NetFileDir* yourself, the value you supply overrides the BDE configuration setting, so... more (see page 1654)
Retrieving Information About a Session (see page 1655)	You can retrieve information about a session and its database components by using a session's informational methods. For example, one method retrieves the names of all aliases known to the session, and another method retrieves the names of tables associated with a specific database component used by the session. The following table summarizes the informational methods to a session component: ***Database-related informational methods for session components***
Closing Database Connections (see page 1656)	To close an individual database connection, call the CloseDatabase method. When you call *CloseDatabase*, the reference count for the database, which is incremented when you call *OpenDatabase*, is decremented by 1. When the reference count for a database is 0, the database is closed. *CloseDatabase* takes one parameter, the database to close. If you opened the database using the *OpenDatabase* method, this parameter can be set to the return value of *OpenDatabase*.
Working with BDE Aliases (see page 1657)	Each database component associated with a session has a BDE alias (although optionally a fully-qualified path name may be substituted for an alias when accessing Paradox and dBASE tables). A session can create, modify, and delete aliases during its lifetime. The AddAlias method creates a new BDE alias for an SQL database server. *AddAlias* takes three parameters: a string containing a name for the alias, a string that specifies the SQL Links driver to use, and a string list populated with parameters for the alias. For example, the following statements use *AddAlias* to add a new alias for accessing an... more (see page 1657)
Naming a Session (see page 1659)	A session's *SessionName* property is used to name the session so that you can associate databases and datasets with it. For the default session, *SessionName* is "Default," For each additional session component you create, you must set its *SessionName* property to a unique value. Database and dataset components have *SessionName* properties that correspond to the *SessionName* property of a session component. If you leave the *SessionName* property blank for a database or dataset component it is automatically associated with the default session. You can also set *SessionName* for a database or dataset component to a name that corresponds to the... more (see page 1659)
Managing Multiple Sessions (see page 1659)	If you create a single application that uses multiple threads to perform database operations, you must create one additional session for each thread. The BDE category on the **Tool palette** contains a session component that you can place in a data module or on a form at design time. **Warning:** When you place a session component, you must also set its *SessionName* property to a unique value so that it does not conflict with the default session's *SessionName* property. Placing a session component at design time presupposes that the number of threads (and therefore sessions) required by the application at... more (see page 1659)
Specifying Default Database Connection Behavior (see page 1661)	KeepConnections provides the default value for the *KeepConnection* property of implicit database components created at runtime. *KeepConnection* specifies what happens to a database connection established for a database component when all its datasets are closed. If *True* (the default), a constant, or *persistent*, database connection is maintained even if no dataset is active. If *False*, a database connection is dropped as soon as all its datasets are closed. **Note:** Connection persistence for a database component you explicitly place in a data module or form is controlled by that database component's *KeepConnection* property. If set differently, *KeepConnection* for a... more (see page 1661)
Dropping Inactive Database Connections (see page 1661)	If the *KeepConnections* property for a session is *True* (the default), then database connections for temporary database components are maintained even if all the datasets used by the component are closed. You can eliminate these connections and free all inactive temporary database components for a session by calling the DropConnections method. For example, the following code frees all inactive, temporary database components for the default session:

Working with Password-protected Paradox and dBASE Tables (see page 1662)	A session component can store passwords for password-protected Paradox and dBASE tables. Once you add a password to the session, your application can open tables protected by that password. Once you remove the password from the session, your application can't open tables that use the password until you add it again.
Managing Database Sessions (see page 1664)	An BDE-based application's database connections, drivers, cursors, queries, and so on are maintained within the context of one or more BDE sessions. Sessions isolate a set of database access operations, such as database connections, without the need to start another instance of the application. All BDE-based database applications automatically include a default session component, named Session, that encapsulates the default BDE session. When database components are added to the application, they are automatically associated with the default session (note that its *SessionName* is "Default"). The default session provides global control over all database components not associated with another session, whether... more (see page 1664)
Managing Database Connections (see page 1665)	You can use a session component to manage the database connections within it. The session component includes properties and methods you can use to • Open database connections (see page 1653). • Close database connections (see page 1656). • Close and free all inactive temporary database connections (see page 1661). • Locate specific database connections (see page 1654). • Iterate through all open database connections (see page 1652).

3.2.3.7.6.1 Activating a Session

Active is a Boolean property that determines if database and dataset components associated with a session are open. You can use this property to read the current state of a session's database and dataset connections, or to change it. If *Active* is *False* (the default), all databases and datasets associated with the session are closed. If *True*, databases and datasets are open.

A session is activated when it is first created, and subsequently, whenever its *Active* property is changed to *True* from *False* (for example, when a database or dataset is associated with a session is opened and there are currently no other open databases or datasets). Setting *Active* to *True* triggers a session's OnStartup event, registers the paradox directory locations. (see page 1654) with the BDE, and registers the *ConfigMode* property, which determines what BDE aliases (see page 1657) are available within the session. You can write an *OnStartup* event handler to initialize the *NetFileDir*, *PrivateDir*, and *ConfigMode* properties before they are registered with the BDE, or to perform other specific session start-up activities.

Once a session is active, you can open its database connections by calling the OpenDatabase method.

For session components you place in a data module or form, setting *Active* to *False* when there are open databases or datasets closes them. At runtime, closing databases and datasets may trigger events associated with them.

Note: You cannot set *Active* to *False* for the default session at design time. While you can close the default session at runtime, it is not recommended.

You can also use a session's *Open* and *Close* methods to activate or deactivate sessions other than the default session at runtime. For example, the following single line of code closes all open databases and datasets for a session:

```
Session1.Close;
Session1->Close();
```

This code sets Session1's *Active* property to *False*. When a session's *Active* property is *False*, any subsequent attempt by the application to open a database or dataset resets *Active* to *True* and calls the session's *OnStartup* event handler if it exists. You can also explicitly code session reactivation at runtime. The following code reactivates *Session1*:

```
Session1.Open;
Session1->Open();
```

Note: If a session is active you can also open and close individual database connections. For more information, see Closing

database connections. (see page 1656)

See Also

Creating Additional Sessions (see page 1652)

Specifying Default Database Connection Behavior (see page 1661)

Managing Database Connections (see page 1665)

Working with BDE Aliases (see page 1657)

Specifying Paradox Directory Locations (see page 1654)

Working with Password Protected Paradox Tables (see page 1662)

Managing Multiple Sessions (see page 1659)

3.2.3.7.6.2 Creating Additional Sessions

You can create sessions to supplement the default session. At design time, you can place additional sessions on a data module (or form), set their properties in the **Object Inspector**, write event handlers for them, and write code that calls their methods. You can also create sessions, set their properties, and call their methods at runtime.

Note: Creating additional sessions is optional unless an application runs concurrent queries against a database or the application is multi-threaded.

To enable dynamic creation of a session component at runtime

1. Declare a *TSession* variable.
2. Instantiate a new session by calling the *Create* method. The constructor sets up an empty list of database components for the session, sets the *KeepConnections* property to *True*, and adds the session to the list of sessions maintained by the application's session list component.
3. Set the *SessionName* property for the new session to a unique name. This property is used to associate database components with the session. For more information about the *SessionName* property, see Naming a session. (see page 1659)
4. Activate the session and optionally adjust its properties.

You can also create and open sessions using the *OpenSession* method of *TSessionList*. Using *OpenSession* is safer than calling *Create*, because *OpenSession* only creates a session if it does not already exist. For information about *OpenSession*, see Managing multiple sessions. (see page 1659).

See Also

Working with BDE Aliases (see page 1657)

Activating a Session (see page 1651)

Specifying Default Database Connection Behavior (see page 1661)

Specifying Paradox Directory Locations (see page 1654)

Managing Database Connections (see page 1665)

Managing Multiple Sessions (see page 1659)

Working with Password Protected Paradox Tables (see page 1662)

3.2.3.7.6.3 Iterating Through a Session's Database Components

You can use two session component properties, *Databases* and *DatabaseCount*, to cycle through all the active database components associated with a session.

Databases is an array of all currently active database components associated with a session. *DatabaseCount* is the number of databases in that array. As connections are opened or closed during a session's life-span, the values of *Databases* and *DatabaseCount* change. For example, if a session's *KeepConnections* property is *False* and all database components are created as needed at runtime, each time a unique database is opened, *DatabaseCount* increases by one. Each time a unique database is closed, *DatabaseCount* decreases by one. If *DatabaseCount* is zero, there are no currently active database components for the session.

The following example code sets the *KeepConnection* property of each active database in the default session to *True*:

```
var
  MaxDbCount: Integer;
begin
  with Session do
    if (DatabaseCount > 0) then
      for MaxDbCount := 0 to (DatabaseCount - 1) do
        Databases[MaxDbCount].KeepConnection := True;
end;
if (Session->DatabaseCount > 0)
  for (int MaxDbCount = 0; MaxDbCount < Session->DatabaseCount; MaxDbCount++)
    Session->Databases[MaxDbCount]->KeepConnection = true;
```

See Also

Opening Database Connections (see page 1653)

Closing Database Connections (see page 1656)

Searching for a Database Connection (see page 1654)

Dropping Inactive Database Connections (see page 1661)

3.2.3.7.6.4 Opening Database Connections

To open a database connection within a session, call the OpenDatabase method. *OpenDatabase* takes one parameter, the name of the database to open. This name is a BDE alias or the name of a database component. For Paradox or dBASE, the name can also be a fully qualified path name. For example, the following statement uses the default session and attempts to open a database connection for the database pointed to by the DBDEMOS alias:

```
var
  DBDemosDatabase: TDatabase;
begin
  DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
  ...
TDatabase *BCDemosDatabase = Session->OpenDatabase("BCDEMOS");
```

OpenDatabase actives the session if it is not already active, and then checks if the specified database name matches the *DatabaseName* property of any database components for the session. If the name does not match an existing database component, *OpenDatabase* creates a temporary database component using the specified name. Finally, *OpenDatabase* calls the *Open* method of the database component to connect to the server. Each call to *OpenDatabase* increments a reference count for the database by 1. As long as this reference count remains greater than 0, the database is open.

See Also

Closing Database Connections (see page 1656)

Dropping Temporary Database Connections (see page 1661)

Iterating Through a Session's Database Components (see page 1652)

Searching for a Database Connection (see page 1654)

3.2.3.7.6.5 Searching for a Database Connection

Use a session's FindDatabase method to determine whether a specified database component is already associated with a session. *FindDatabase* takes one parameter, the name of the database to search for. This name is a BDE alias or database component name. For Paradox or dBASE, it can also be a fully-qualified path name.

FindDatabase returns the database component if it finds a match. Otherwise it returns **nil**.

The following code searches the default session for a database component using the DBDEMOS alias, and if it is not found, creates one and opens it:

```
var
  DB: TDatabase;
begin
  DB := Session.FindDatabase('DBDEMOS');
  if (DB = nil) then                          { database doesn't exist for session so,}
    DB := Session.OpenDatabase('DBDEMOS');    { create and open it}
  if Assigned(DB) and DB.Connected then begin
    DB.StartTransaction;
    ...
  end;
end;
```

```
TDatabase *DB = Session->FindDatabase("BCDEMOS");
if ( !DB )                                  // Database does not exist for session so
  DB = Session->OpenDatabase("BCDEMOS"); // create and open it
if (DB && DB->Connected)
{
  if (!DB->InTransaction)
  {
    DB->StartTransaction();
    .
    .
    .
  }
}
```

See Also

Opening Database Connections (see page 1653)

Closing Database Connections (see page 1656)

Iterating Through a Session's Database Components (see page 1652)

Dropping Inactive Database Connections (see page 1661)

3.2.3.7.6.6 Specifying Paradox Directory Locations

Two session component properties, *NetFileDir* and *PrivateDir*, are specific to applications that work with Paradox tables.

NetFileDir specifies the directory that contains the Paradox network control file, PDOXUSRS.NET. This file governs sharing of Paradox tables on network drives. All applications that need to share Paradox tables must specify the same directory for the network control file (typically a directory on a network file server). Delphi derives a value for *NetFileDir* from the Borland Database Engine (BDE) configuration file for a given database alias. If you set *NetFileDir* yourself, the value you supply overrides the BDE configuration setting, so be sure to validate the new value.

At design time, you can specify a value for *NetFileDir* in the **Object Inspector**. You can also set or change *NetFileDir* in code at runtime. The following code sets *NetFileDir* for the default session to the location of the directory from which your application runs:

```
Session.NetFileDir := ExtractFilePath(Application.EXEName);
```

```
Session->NetFileDir = ExtractFilePath(ParamStr(0));
```

Note: *NetFileDir* can only be changed when an application does not have any open Paradox files. If you change *NetFileDir* at runtime, verify that it points to a valid network directory that is shared by your network users.

PrivateDir specifies the directory for storing temporary table processing files, such as those generated by the BDE to handle local SQL statements. If no value is specified for the *PrivateDir* property, the BDE automatically uses the current directory at the time it is initialized. If your application runs directly from a network file server, you can improve application performance at runtime by setting *PrivateDir* to a user's local hard drive before opening the database.

Note: Do not set *PrivateDir* at design time and then open the session in the IDE. Doing so generates a Directory is busy error when running your application from the IDE.

The following code changes the setting of the default session's *PrivateDir* property to a user's C:\TEMP directory:

```
Session.PrivateDir := "C:\TEMP";
Session->PrivateDir = "C:\\TEMP";
```

Warning: Do not set *PrivateDir* to a root directory on a drive. Always specify a subdirectory.

See Also

Creating Additional Sessions (see page 1652)

Activating a Session (see page 1651)

Specifying Default Database Connection Behavior (see page 1661)

Working with BDE Aliases (see page 1657)

Managing Database Connections (see page 1665)

Managing Multiple Sessions (see page 1659)

Working with Password Protected Paradox Tables (see page 1662)

3.2.3.7.6.7 Retrieving Information About a Session

You can retrieve information about a session and its database components by using a session's informational methods. For example, one method retrieves the names of all aliases known to the session, and another method retrieves the names of tables associated with a specific database component used by the session. The following table summarizes the informational methods to a session component:

Database-related informational methods for session components

Method	Purpose
GetAliasDriverName	Retrieves the BDE driver for a specified alias of a database.
GetAliasNames	Retrieves the list of BDE aliases for a database.
GetAliasParams	Retrieves the list of parameters for a specified BDE alias of a database.
GetConfigParams	Retrieves configuration information from the BDE configuration file.
GetDatabaseNames	Retrieves the list of BDE aliases and the names of any *TDatabase* components currently in use.
GetDriverNames	Retrieves the names of all currently installed BDE drivers.
GetDriverParams	Retrieves the list of parameters for a specified BDE driver.
GetStoredProcNames	Retrieves the names of all stored procedures for a specified database.
GetTableNames	Retrieves the names of all tables matching a specified pattern for a specified database.
GetFieldNames	Retrieves the names of all fields in a specified table in a specified database.

Except for *GetAliasDriverName*, these methods return a set of values into a string list declared and maintained by your

application. (*GetAliasDriverName* returns a single string, the name of the current BDE driver for a particular database component used by the session.)

For example, the following code retrieves the names of all database components and aliases known to the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    Session.GetDatabaseNames(List);
    ...
  finally
    List.Free;
  end;
end;
```

```
TStringList *List = new TStringList();
try
{
  Session->GetDatabaseNames(List);
  .
  .
  .
}
catch (...)
{
delete List;
throw;
}
delete List;
```

See Also

Working with BDE Aliases (see page 1657)

Managing Database Connections (see page 1665)

3.2.3.7.6.8 Closing Database Connections

To close an individual database connection, call the CloseDatabase method. When you call *CloseDatabase*, the reference count for the database, which is incremented when you call *OpenDatabase*, is decremented by 1. When the reference count for a database is 0, the database is closed. *CloseDatabase* takes one parameter, the database to close. If you opened the database using the *OpenDatabase* method, this parameter can be set to the return value of *OpenDatabase*.

```
Session.CloseDatabase(DBDemosDatabase);
Session->CloseDatabase(BCDemosDatabase);
```

If the specified database name is associated with a temporary (implicit) database component, and the session's *KeepConnections* property is *False*, the database component is freed, effectively closing the connection.

Note: If *KeepConnections* is *False* temporary database components are closed and freed automatically when the last dataset associated with the database component is closed. An application can always call *CloseDatabase* prior to that time to force closure. To free temporary database components when *KeepConnections* is *True*, call the database component's *Close* method, and then call the session's *DropConnections* method.

Note: Calling *CloseDatabase* for a persistent database component does not actually close the connection. To close the connection, call the database component's *Close* method directly.

There are two ways to close all database connections within the session:

- Set the *Active* property for the session to *False*.

- Call the *Close* method for the session.

When you set *Active* to *False*, Delphi automatically calls the *Close* method. *Close* disconnects from all active databases by freeing temporary database components and calling each persistent database component's *Close* method. Finally, *Close* sets the session's BDE handle to **nil**.

See Also

Opening Database Connections (see page 1653)

Dropping Temporary Database Connections (see page 1661)

Iterating Through a Session's Database Components (see page 1652)

Searching for a Database Connection (see page 1654)

3.2.3.7.6.9 Working with BDE Aliases

Each database component associated with a session has a BDE alias (although optionally a fully-qualified path name may be substituted for an alias when accessing Paradox and dBASE tables). A session can create, modify, and delete aliases during its lifetime.

The AddAlias method creates a new BDE alias for an SQL database server. *AddAlias* takes three parameters: a string containing a name for the alias, a string that specifies the SQL Links driver to use, and a string list populated with parameters for the alias. For example, the following statements use *AddAlias* to add a new alias for accessing an InterBase server to the default session:

```
var
  AliasParams: TStringList;
begin
  AliasParams := TStringList.Create;
  try
    with AliasParams do begin
      Add('OPEN MODE=READ');
      Add('USER NAME=TOMSTOPPARD');
      Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
    end;
    Session.AddAlias('CATS', 'INTRBASE', AliasParams);
    ...
  finally
    AliasParams.Free;
  end;
end;
```

```
TStringList *AliasParams = new TStringList();
try
{
  AliasParams->Add("OPEN MODE=READ");
  AliasParams->Add("USER NAME=TOMSTOPPARD");
  AliasParams->Add("SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB");
  Session->AddAlias("CATS", "INTRBASE", AliasParams);
  .
  .
  .
}
catch (...)
{
  delete AliasParams;
  throw;
}
delete AliasParams;
```

AddStandardAlias creates a new BDE alias for Paradox, dBASE, or ASCII tables. *AddStandardAlias* takes three string parameters: the name for the alias, the fully-qualified path to the Paradox or dBASE table to access, and the name of the default driver to use when attempting to open a table that does not have an extension. For example, the following statement uses

AddStandardAlias to create a new alias for accessing a Paradox table:

```
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
Session->AddStandardAlias("MYBCDEMOS", "C:\\TESTING\\DEMOS\\", "Paradox");
```

When you add an alias to a session, the BDE stores a copy of the alias in memory, where it is only available to this session and any other sessions with *cfmPersistent* included in the *ConfigMode* property. *ConfigMode* is a set that describes which types of aliases can be used by the databases in the session. The default setting is *cmAll*, which translates into the set [*cfmVirtual*, *cfmPersistent, cfmSession*]. If *ConfigMode* is *cmAll*, a session can see all aliases created within the session (*cfmSession*), all aliases in the BDE configuration file on a user's system (*cfmPersistent*), and all aliases that the BDE maintains in memory (*cfmVirtual*). You can change *ConfigMode* to restrict what BDE aliases the databases in a session can use. For example, setting *ConfigMode* to *cfmSession* restricts a session's view of aliases to those created within the session. All other aliases in the BDE configuration file and in memory are not available.

To make a newly created alias available to all sessions and to other applications, use the session's SaveConfigFile method. *SaveConfigFile* writes aliases in memory to the BDE configuration file where they can be read and used by other BDE-enabled applications.

After you create an alias, you can make changes to its parameters by calling ModifyAlias. *ModifyAlias* takes two parameters: the name of the alias to modify and a string list containing the parameters to change and their values. For example, the following statements use *ModifyAlias* to change the OPEN MODE parameter for the CATS alias to READ/WRITE in the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  with List do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
  end;
  Session.ModifyAlias('CATS', List);
  List.Free;
  ...
TStringList *List = new TStringList();
List->Clear();
List->Add("OPEN MODE=READ/WRITE");
Session->ModifyAlias("CATS", List);
delete List;
```

To delete an alias previously created in a session, call the DeleteAlias method. *DeleteAlias* takes one parameter, the name of the alias to delete. *DeleteAlias* makes an alias unavailable to the session.

Note: *DeleteAlias* does not remove an alias from the BDE configuration file if the alias was written to the file by a previous call to *SaveConfigFile*. To remove the alias from the configuration file after calling *DeleteAlias*, call *SaveConfigFile* again.

Session components provide five methods for retrieving information about a BDE aliases, including parameter information and driver information. They are:

- GetAliasNames, to list the aliases to which a session has access.
- GetAliasParams, to list the parameters for a specified alias.
- GetAliasDriverName, to return the name of the BDE driver used by the alias.
- GetDriverNames, to return a list of all BDE drivers available to the session.
- GetDriverParams, to return driver parameters for a specified driver.

For more information about using a session's informational methods, see Using transactions with the BDE. (see page 1655).
For more information about BDE aliases see the BDE online help, BDE32.HLP.

See Also

Creating Additional Sessions (see page 1652)

Activating a Session (see page 1651)

Specifying Default Database Connection Behavior (see page 1661)

Specifying Paradox Directory Locations (see page 1654)

Managing Database Connections (see page 1665)

Managing Multiple Sessions (see page 1659)

Working with Password Protected Paradox Tables (see page 1662)

3.2.3.7.6.10 Naming a Session

A session's *SessionName* property is used to name the session so that you can associate databases and datasets with it. For the default session, *SessionName* is "Default," For each additional session component you create, you must set its *SessionName* property to a unique value.

Database and dataset components have *SessionName* properties that correspond to the *SessionName* property of a session component. If you leave the *SessionName* property blank for a database or dataset component it is automatically associated with the default session. You can also set *SessionName* for a database or dataset component to a name that corresponds to the *SessionName* of a session component you create.

The following code uses the *OpenSession* method of the default *TSessionList* component, *Sessions*, to open a new session component, sets its *SessionName* to "InterBaseSession," activate the session, and associate an existing database component *Database1* with that session:

```
var
  IBSession: TSession;
  ...
begin
  IBSession := Sessions.OpenSession('InterBaseSession');
  Database1.SessionName := 'InterBaseSession';
end;
```

```
TSession *IBSession = Sessions->OpenSession("InterBaseSession");
Database1->SessionName = "InterBaseSession";
```

For more information about using *Sessions*, see Managing Multiple Sessions. (see page 1659).

See Also

Creating Additional Sessions (see page 1652)

3.2.3.7.6.11 Managing Multiple Sessions

If you create a single application that uses multiple threads to perform database operations, you must create one additional session for each thread. The BDE category on the **Tool palette** contains a session component that you can place in a data module or on a form at design time.

Warning: When you place a session component, you must also set its *SessionName* property to a unique value so that it does not conflict with the default session's *SessionName* property.

Placing a session component at design time presupposes that the number of threads (and therefore sessions) required by the application at runtime is static. More likely, however, is that an application needs to create sessions dynamically. To create sessions dynamically, call the OpenSession method of the global Sessions object at runtime.

OpenSession requires a single parameter, a name for the session that is unique across all session names for the application. The following code dynamically creates and activates a new session with a uniquely generated name:

```
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

```
Sessions->OpenSession("RunTimeSession" + IntToStr(Sessions->Count + 1));
```

This statement generates a unique name for a new session by retrieving the current number of sessions, and adding one to that value. Note that if you dynamically create and destroy sessions at runtime, this example code will not work as expected. Nevertheless, this example illustrates how to use the properties and methods of *Sessions* to manage multiple sessions.

Sessions is a variable of type *TSessionList* that is automatically instantiated for BDE-based database applications. You use the properties and methods of *Sessions* to keep track of multiple sessions in a multi-threaded database application. The following table summarizes the properties and methods of the *TSessionList* component:

TSessionList properties and methods

Property or Method	Purpose
Count	Returns the number of sessions, both active and inactive, in the session list.
FindSession	Searches for a session with a specified name and returns a pointer to it, or **nil** if there is no session with the specified name. If passed a blank session name, *FindSession* returns a pointer to the default session, *Session*.
GetSessionNames	Populates a string list with the names of all currently instantiated session components. This procedure always adds at least one string, "Default" for the default session.
List	Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised.
OpenSession	Creates and activates a new session or reactivates an existing session for a specified session name.
Sessions	Accesses the session list by ordinal value.

As an example of using *Sessions* properties and methods in a multi-threaded application, consider what happens when you want to open a database connection. To determine if a connection already exists, use the *Sessions* property to walk through each session in the sessions list, starting with the default session. For each session component, examine its *Databases* property to see if the database in question is open. If you discover that another thread is already using the desired database, examine the next session in the list.

If an existing thread is not using the database, then you can open the connection within that session.

If, on the other hand, all existing threads are using the database, you must open a new session in which to open another database connection.

If you are replicating a data module that contains a session in a multi-threaded application, where each thread contains its own copy of the data module, you can use the AutoSessionName property to make sure that all datasets in the data module use the correct session. Setting *AutoSessionName* to *True* causes the session to generate its own unique name dynamically when it is created at runtime. It then assigns this name to every dataset in the data module, overriding any explicitly set session names. This ensures that each thread has its own session, and each dataset uses the session in its own data module.

See Also

Working with BDE Aliases (see page 1657)

Activating a Session (see page 1651)

Specifying Default Database Connection Behavior (see page 1661)

Specifying Paradox Directory Locations (see page 1654)

Managing Database Connections (see page 1665)

Creating Additional Sessions (see page 1652)

Working with Password Protected Paradox Tables (see page 1662)

3.2.3.7.6.12 Specifying Default Database Connection Behavior

KeepConnections provides the default value for the *KeepConnection* property of implicit database components created at runtime. *KeepConnection* specifies what happens to a database connection established for a database component when all its datasets are closed. If *True* (the default), a constant, or *persistent*, database connection is maintained even if no dataset is active. If *False*, a database connection is dropped as soon as all its datasets are closed.

Note: Connection persistence for a database component you explicitly place in a data module or form is controlled by that database component's *KeepConnection* property. If set differently, *KeepConnection* for a database component always overrides the *KeepConnections* property of the session. For more information about controlling individual database connections within a session, see Managing database connections. (see page 1665)

KeepConnections should be set to *True* for applications that frequently open and close all datasets associated with a database on a remote server. This setting reduces network traffic and speeds data access because it means that a connection need only be opened and closed once during the lifetime of the session. Otherwise, every time the application closes or reestablishes a connection, it incurs the overhead of attaching and detaching the database.

Note: Even when *KeepConnections* is *True* for a session, you can close and free inactive database connections for all implicit database components by calling the DropConnections method. For more information about *DropConnections*, see Dropping inactive database connections (see page 1661).

See Also

Creating Additional Sessions (see page 1652)

Activating a Session (see page 1651)

Managing Database Connections (see page 1665)

Working with BDE Aliases (see page 1657)

Specifying Paradox Directory Locations (see page 1654)

Working with Password Protected Paradox Tables (see page 1662)

Managing Multiple Sessions (see page 1659)

3.2.3.7.6.13 Dropping Inactive Database Connections

If the *KeepConnections* property for a session is *True* (the default), then database connections for temporary database components are maintained even if all the datasets used by the component are closed. You can eliminate these connections and free all inactive temporary database components for a session by calling the DropConnections method. For example, the following code frees all inactive, temporary database components for the default session:

```
Session.DropConnections;
Session->DropConnections();
```

Temporary database components for which one or more datasets are active are not dropped or freed by this call. To free these components, call Close.

See Also

Opening Database Connections (see page 1653)

Closing Database Connections (see page 1656)

Iterating Through a Session's Database Components (see page 1652)

Searching for a Database Connection (see page 1654)

3.2.3.7.6.14 Working with Password-protected Paradox and dBASE Tables

A session component can store passwords for password-protected Paradox and dBASE tables. Once you add a password to the session, your application can open tables protected by that password. Once you remove the password from the session, your application can't open tables that use the password until you add it again.

Using the AddPassword method

The AddPassword method provides an optional way for an application to provide a password for a session prior to opening an encrypted Paradox or dBASE table that requires a password for access. If you do not add the password to the session, when your application attempts to open a password-protected table, a dialog box prompts the user for a password.

AddPassword takes one parameter, a string containing the password to use. You can call *AddPassword* as many times as necessary to add passwords (one at a time) to access tables protected with different passwords.

```
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Session.AddPassword(Passwrd);
  try
    Table1.Open;
  except
    ShowMessage('Could not open table!');
    Application.Terminate;
  end;
end;
```

```
AnsiString PassWrd;
PassWrd = InputBox("Enter password", "Password:", "");
Session->AddPassword(PassWrd);
try
{
  Table1->Open();
}
catch(...)
{
  ShowMessage("Could not open table!");
  Application->Terminate();
}
```

Note: Use of the *InputBox* function, above, is for demonstration purposes. In a real-world application, use password entry facilities that mask the password as it is entered, such as the *PasswordDialog* function or a custom form.

The Add button of the PasswordDialog function dialog has the same effect as the *AddPassword* method.

```
if PasswordDialog(Session) then
  Table1.Open
else
  ShowMessage('No password given, could not open table!');
end;
```

```
if (PasswordDlg(Session))
  Table1->Open();
else
  ShowMessage("No password given, could not open table!");
```

Using the RemovePassword and RemoveAllPasswords methods

RemovePassword deletes a previously added password from memory. *RemovePassword* takes one parameter, a string containing the password to delete.

```
Session.RemovePassword('secret');
```

```
Session->RemovePassword("secret");
```

RemoveAllPasswords deletes all previously added passwords from memory.

```
Session.RemoveAllPasswords;
Session->RemoveAllPasswords();
```

Using the GetPassword method and OnPassword event

The OnPassword event allows you to control how your application supplies passwords for Paradox and dBASE tables when they are required. Provide a handler for the OnPassword event if you want to override the default password handling behavior. If you do not provide a handler, Delphi presents a default dialog for entering a password and no special behavior is provided—the table open attempt either succeeds or an exception is raised.

If you provide a handler for the OnPassword event, do two things in the event handler: call the AddPassword method and set the event handler's Continue parameter to True. The AddPassword method passes a string to the session to be used as a password for the table. The Continue parameter indicates to Delphi that no further password prompting need be done for this table open attempt. The default value for Continue is False, and so requires explicitly setting it to True. If Continue is False after the event handler has finished executing, an OnPassword event fires again—even if a valid password has been passed using AddPassword. If Continue is True after execution of the event handler and the string passed with AddPassword is not the valid password, the table open attempt fails and an exception is raised.

OnPassword can be triggered by two circumstances. The first is an attempt to open a password-protected table (dBASE or Paradox) when a valid password has not already been supplied to the session. (If a valid password for that table has already been supplied, the OnPassword event does not occur.)

The other circumstance is a call to the GetPassword method. GetPassword either generates an OnPassword event, or, if the session does not have an OnPassword event handler, displays a default password dialog. It returns True if the OnPassword event handler or default dialog added a password to the session, and False if no entry at all was made.

In the following example, the Password method is designated as the OnPassword event handler for the default sessionby assigning it to the global Session object's OnPassword property.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.OnPassword := Password;
end;
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Session->OnPassword = Password;
}
```

In the *Password* method, the *InputBox* function prompts the user for a password. The *AddPassword* method then programmatically supplies the password entered in the dialog to the session.

```
procedure TForm1.Password(Sender: TObject; var Continue: Boolean);
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Continue := (Passwrd > '');
  Session.AddPassword(Passwrd);
end;
void __fastcall TForm1::Password(TObject *Sender, bool &Continue)
{
  AnsiString PassWrd = InputBox("Enter password", "Password:", "");
  Session->AddPassword(PassWrd);
  Continue = (PassWrd > "");
}
```

The *OnPassword* event (and thus the *Password* event handler) is triggered by an attempt to open a password-protected table, as demonstrated below. Even though the user is prompted for a password in the handler for the *OnPassword* event, the table open attempt can still fail if they enter an invalid password or something else goes wrong.

```
procedure TForm1.OpenTableBtnClick(Sender: TObject);
const
CRLF = #13 + #10;
```

```
begin
  try
    Table1.Open;                                { this line triggers the OnPassword event }
  except
    on E:Exception do begin                             { exception if cannot open table }
      ShowMessage('Error!' + CRLF +              { display error explaining what happened }
        E.Message + CRLF +
        'Terminating application...');
      Application.Terminate;                                      { end the application }
    end;
  end;
end;
void __fastcall TForm1::OpenTableBtnClick(TObject *Sender)
{
  try
  {
    // this line triggers the OnPassword event
    Table1->Open();
  }
  // exception if cannot open table
  catch(...)
  {
    ShowMessage("Could not open table!");
    Application->Terminate();
  }
}
```

See Also

Creating Additional Sessions (see page 1652)

Activating a Session (see page 1651)

Specifying Default Database Connection Behavior (see page 1661)

Working with BDE Aliases (see page 1657)

Specifying Paradox Directory Locations (see page 1654)

Managing Database Connections (see page 1665)

Managing Multiple Sessions (see page 1659)

3.2.3.7.6.15 Managing Database Sessions

An BDE-based application's database connections, drivers, cursors, queries, and so on are maintained within the context of one or more BDE sessions. Sessions isolate a set of database access operations, such as database connections, without the need to start another instance of the application.

All BDE-based database applications automatically include a default session component, named Session, that encapsulates the default BDE session. When database components are added to the application, they are automatically associated with the default session (note that its *SessionName* is "Default"). The default session provides global control over all database components not associated with another session, whether they are implicit (created by the session at runtime when you open a dataset that is not associated with a database component you create) or persistent (explicitly created by your application). The default session is not visible in your data module or form at design time, but you can access its properties and methods in your code at runtime.

To use the default session, you need write no code unless your application must

- Explicitly activate or deactivate a session (see page 1651), enabling or disabling the session's databases' ability to open.
- Modify the properties of the session, such as specifying default properties for implicitly generated database components (see page 1661).
- Execute a session's methods, such as managing database connections (see page 1665) (for example opening and closing

database connections in response to user actions).

- Respond to session events, such as when the application attempts to access a password-protected Paradox or dBASE table (see page 1662).
- Set Paradox directory locations (see page 1654) such as the *NetFileDir* property to access Paradox tables on a network and the *PrivateDir* property to a local hard drive to speed performance.
- Manage the BDE aliases (see page 1657) that describe possible database connection configurations for databases and datasets that use the session.

Whether you add database components to an application at design time or create them dynamically at runtime, they are automatically associated with the default session unless you specifically assign them to a different session (see page 1645). If you open a dataset that is not associated with a database component, Delphi automatically

- Creates a database component for it at runtime.
- Associates the database component with the default session.
- Initializes some of the database component's key properties based on the default session's properties. Among the most important of these properties is *KeepConnections*, which determines when database connections are maintained or dropped by an application.

The default session provides a widely applicable set of defaults that can be used as is by most applications. You need only associate a database component with an explicitly named session if the component performs a simultaneous query against a database already opened by the default session. In this case, each concurrent query must run under its own session. Multi-threaded database applications also require multiple sessions, where each thread has its own session.

Applications can create additional session components (see page 1652) as needed. BDE-based database applications automatically include a session list component, named Sessions, that you can use to manage all of your session components. For more information about managing multiple sessions see, Managing multiple sessions. (see page 1659)

You can safely place session components in data modules. If you put a data module that contains one or more session components into the Object Repository, however, make sure to set the *AutoSessionName* property to *True* to avoid namespace conflicts when users inherit from it.

See Also

Using BDE Enabled Datasets (see page 1675)

Connecting to Databases with TDatabase (see page 1646)

3.2.3.7.6.16 Managing Database Connections

You can use a session component to manage the database connections within it. The session component includes properties and methods you can use to

- Open database connections (see page 1653).
- Close database connections (see page 1656).
- Close and free all inactive temporary database connections (see page 1661).
- Locate specific database connections (see page 1654).
- Iterate through all open database connections (see page 1652).

See Also

Creating Additional Sessions (see page 1652)

Activating a Session (see page 1651)

Specifying Default Database Connection Behavior (see page 1661)

Working with BDE Aliases (see page 1657)

Specifying Paradox Directory Locations (see page 1654)

Working with Password Protected Paradox Tables (see page 1662)

Managing Multiple Sessions (see page 1659)

3.2.3.7.7 Using BDE-enabled datasets

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Topics

Name	Description
Associating a Dataset with Database and Session Connections (see page 1668)	In order for a BDE-enabled dataset to fetch data from a database server it needs to use both a database and a session. Databases (see page 1646) represent connections to specific database servers. The database identifies a BDE driver, a particular database server that uses that driver, and a set of connection parameters for connecting to that database server. Each database is represented by a TDatabase component. You can either associate your datasets with a *TDatabase* component you add to a form or data module, or you can simply identify the database server by name and let Delphi generate an implicit database component... more (see page 1668)
Binding Parameters (see page 1669)	When you prepare and execute a stored procedure, its input parameters are automatically bound to parameters on the server. *TStoredProc* lets you use the ParamBindMode property to specify how parameters should be bound to the parameters on the server. By default *ParamBindMode* is set to *pbByName*, meaning that parameters from the stored procedure component are matched to those on the server by name. This is the easiest method of binding parameters. Some servers also support binding parameters by ordinal value, the order in which the parameters appear in the stored procedure. In this case the order in which you... more (see page 1669)
Caching BLOBs (see page 1670)	BDE-enabled datasets all have a CacheBlobs property that controls whether BLOB fields are cached locally by the BDE when an application reads BLOB records. By default, *CacheBlobs* is *True*, meaning that the BDE caches a local copy of BLOB fields. Caching BLOBs improves application performance by enabling the BDE to store local copies of BLOBs instead of fetching them repeatedly from the database server as a user scrolls through records. In applications and environments where BLOBs are frequently updated or replaced, and a fresh view of BLOB data is more important than application performance, you can set *CacheBlobs* to... more (see page 1670)
Controlling Read/Write Access to Local Tables (see page 1670)	Like any table type dataset, *TTable* lets you control read and write access (see page 1587) by your application using the ReadOnly property. In addition, for Paradox, dBASE, and FoxPro tables, *TTable* can let you control read and write access to tables by other applications. The Exclusive property controls whether your application gains sole read/write access to a Paradox, dBASE, or FoxPro table. To gain sole read/write access for these table types, set the table component's *Exclusive* property to *True* before opening the table. If you succeed in opening a table for exclusive access, other applications cannot read data from or write data... more (see page 1670)
Importing Data from Another Table (see page 1670)	You can use a table component's BatchMove*method to import data from another table. BatchMove can* • Copy records from another table into this table. • Update records in this table that occur in another table. • Append records from another table to the end of this table. • Delete records in this table that occur in another table. *BatchMove* takes two parameters: the name of the table from which to import data, and a mode specification that determines which import operation to perform. The following table describes the possible settings for the mode specification: ***BatchMove import modes***

Working with BDE Handle Properties (see page 1671)	You can use BDE-enabled datasets without ever needing to make direct API calls to the Borland Database Engine. The BDE-enabled datasets, in combination with database and session components, encapsulate much of the BDE functionality. However, if you need to make direct API calls to the BDE, you may need BDE handles for resources managed by the BDE. Many BDE APIs require these handles as parameters. All BDE-enabled datasets include three read-only properties for accessing BDE handles at runtime: • Handle is a handle to the BDE cursor that accesses the records in the dataset. • DBHandle is a handle to the... more (see page 1671)
Renaming a Table (see page 1671)	To rename a Paradox or dBASE table at runtime, call the table's RenameTable method. For example, the following statement renames the Customer table to CustInfo:
Specifying a dBASE Index File (see page 1672)	For most servers, you use the methods common to all table type datasets to specify an index. These methods are described in Sorting records with indexes (see page 1609). For dBASE tables that use non-production index files or dBASE III PLUS-style indexes (*.NDX), however, you must use the *IndexFiles and IndexNameproperties instead. Set the IndexFiles property to the name of the non-production index file or list the .NDX files. Then, specify one index in the IndexName property to have it actively sorting the dataset.* *At design time, click the ellipsis button in the IndexFiles property value in the **Object Inspector** to... more (see page 1672)*
Specifying the Table Type for Local Tables (see page 1673)	If an application accesses Paradox, dBASE, FoxPro, or comma-delimited ASCII text tables, then the BDE uses the TableType property to determine the table's type (its expected structure). *TableType* is not used when *TTable* represents an SQL-based table on a database server. By default *TableType* is set to *ttDefault*. When *TableType* is *ttDefault*, the BDE determines a table's type from its filename extension. The following table summarizes the file extensions recognized by the BDE and the assumptions it makes about a table's type: ***Table types recognized by the BDE based on file extension***
Using TQuery (see page 1674)	*TQuery* represents a single Data Definition Language (DDL) or Data Manipulation Language (DML) statement (For example, a SELECT, INSERT, DELETE, UPDATE, CREATE INDEX, or ALTER TABLE command). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. *TQuery* implements all of the basic functionality introduced by TDataSet (see page 1632), as well as all of the special features typical of query-type datasets (see page 1618). Because *TQuery* is a BDE-enabled dataset, it must usually be associated with a database and a session (see page 1668). (The one exception is when you use the *TQuery* for a heterogeneous query.)... more (see page 1674)
Using TStoredProc (see page 1674)	*TStoredProc* represents a stored procedure. It implements all of the basic functionality introduced by TDataSet (see page 1632), as well as most of the special features typical of stored procedure-type datasets (see page 1618). Because *TStoredProc* is a BDE-enabled dataset, it must be associated with a database and a session (see page 1668). Once the dataset is associated with a database and session, you can bind it to a particular stored procedure by setting the StoredProcName property. *TStoredProc* differs from other stored procedure-type datasets in the following ways: • It gives you greater control over how to bind parameters (see page 1669). • It provides support for Oracle overloaded stored (see page 1677)... more (see page 1674)
Using TTable (see page 1674)	*TTable* encapsulates the full structure of and data in an underlying database table. It implements all of the basic functionality introduced by TDataSet (see page 1632), as well as all of the special features typical of table type datasets (see page 1620). Because *TTable* is a BDE-enabled dataset, it must be associated with a database and a session (see page 1668). Once the dataset is associated with a database and session, you can bind it to a particular database table by setting the TableName property and, if you are using a Paradox, dBASE, FoxPro, or comma-delimited ASCII text table, the TableType property. **Note:** The table must... more (see page 1674)

Using BDE-enabled Datasets (see page 1675)	BDE-enabled datasets use the Borland Database Engine (BDE) to access data. They inherit the common dataset capabilities described in Understanding datasets (see page 1632), using the BDE to provide the implementation. In addition, all BDE datasets add properties, events, and methods for • Associating a dataset with database and session connections (see page 1668). • Caching BLOBs (see page 1670). • Obtaining a BDE handle (see page 1671). There are three BDE-enabled datasets: • *TTable*, a table type dataset (see page 1620) that represents all of the rows and columns of a single database table. See Using TTable (see page 1674) for a description of features unique to *TTable*. • *TQuery*, a query-type dataset (see page 1618) that... more (see page 1675)
Creating Heterogenous Queries (see page 1676)	*TQuery* supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table. When you execute a heterogeneous query, the BDE parses and processes the query using Local SQL. Because BDE uses Local SQL, extended, server-specific SQL syntax is not supported.
Obtaining an Editable Result Set (see page 1676)	To request a result set that users can edit in data-aware controls, set a query component's RequestLive property to True. Setting *RequestLive* to True does not guarantee a live result set, but the BDE attempts to honor the request whenever possible. There are some restrictions on live result set requests, depending on whether the query uses the local SQL parser or a server's SQL parser. • Queries where table names are preceded by a BDE database alias (as in heterogeneous queries) and queries executed against Paradox or dBASE are parsed by the BDE using Local SQL. When queries use the local... more (see page 1676)
Updating a Read-only Result Set (see page 1677)	Applications can update data returned in a read-only result set if they are using cached updates. If you are using a client dataset to cache updates (see page 1731), the client dataset or its associated provider can automatically generate the SQL for applying updates unless the query represents multiple tables. If the query represents multiple tables, you must indicate how to apply the updates: If all updates are applied to a single database table, you can indicate the underlying table to update in an OnGetTableName event handler. If you need more control over applying updates, you can associate the query with an... more (see page 1677)
Working with Oracle Overloaded Stored Procedures (see page 1677)	Oracle servers allow overloading of stored procedures; overloaded procedures are different procedures with the same name. The stored procedure component's Overload property enables an application to specify the procedure to execute. If *Overload* is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then the stored procedure component executes the first stored procedure it finds on the Oracle server that has the overloaded name; if it is two (2), it executes the second, and so on. **Note:** Overloaded stored procedures may take different input and output parameters. See your Oracle server documentation for more... more (see page 1677)

3.2.3.7.7.1 Associating a Dataset with Database and Session Connections

In order for a BDE-enabled dataset to fetch data from a database server it needs to use both a database and a session.

Databases (see page 1646) represent connections to specific database servers. The database identifies a BDE driver, a particular database server that uses that driver, and a set of connection parameters for connecting to that database server. Each database is represented by a TDatabase component. You can either associate your datasets with a *TDatabase* component you add to a form or data module, or you can simply identify the database server by name and let Delphi generate an implicit database component for you. Using an explicitly-created *TDatabase* component is recommended for most applications, because

the database component gives you greater control over how the connection is established, including the login process, and lets you create and use transactions.

To associate a BDE-enabled dataset with a database, use the *DatabaseName property. DatabaseName is a string that contains different information, depending on whether you are using an explicit database component and, if not, the type of database you are using:*

- If you are using an explicit *TDatabase* component, *DatabaseName* is the value of the *DatabaseName* property of the database component.
- If you want to use an implicit database component and the database has a BDE alias, you can specify a BDE alias as the value of *DatabaseName*. A BDE alias represents a database plus configuration information for that database. The configuration information associated with an alias differs by database type (Oracle, Sybase, InterBase, Paradox, dBASE, and so on).
- If you want to use an implicit database component for a Paradox or dBASE database, you can also use *DatabaseName* to simply specify the directory where the database tables are located.

A session (see page 1664) provides global management for a group of database connections in an application. When you add BDE-enabled datasets to your application, your application automatically contains a session component, named *Session. As you add database and dataset components to the application, they are automatically associated with this default session. It also controls access to password protected Paradox files, and it specifies directory locations for sharing Paradox files over a network. You can control database connections and access to Paradox files using the properties, events, and methods of the session.*

You can use the default session to control all database connections in your application. Alternatively, you can add additional session components at design time or create them dynamically at runtime to control a subset of database connections in an application. To associate your dataset with an explicitly created session component, use the SessionName property. If you do not use explicit session components in your application, you do not have to provide a value for this property. Whether you use the default session or explicitly specify a session using the SessionName property, you can access the session associated with a dataset by reading the DBSession property.

***Note:** If you use a session component, the SessionName property of a dataset must match the SessionName property for the database component with which the dataset is associated.*

See Also

Caching BLOBs (see page 1670)

Working with BDE Handle Properties (see page 1671)

3.2.3.7.7.2 Binding Parameters

When you prepare and execute a stored procedure, its input parameters are automatically bound to parameters on the server.

TStoredProc lets you use the ParamBindMode property to specify how parameters should be bound to the parameters on the server. By default *ParamBindMode* is set to *pbByName*, meaning that parameters from the stored procedure component are matched to those on the server by name. This is the easiest method of binding parameters.

Some servers also support binding parameters by ordinal value, the order in which the parameters appear in the stored procedure. In this case the order in which you specify parameters in the parameter collection editor is significant. The first parameter you specify is matched to the first input parameter on the server, the second parameter is matched to the second input parameter on the server, and so on. If your server supports parameter binding by ordinal value, you can set *ParamBindMode* to *pbByNumber*.

Tip: If you want to set *ParamBindMode* to *pbByNumber*, you need to specify the correct parameter types in the correct order.

See Also

Working with Oracle Overloaded Stored Procedures (see page 1677)

3.2.3.7.7.3 Caching BLOBs

BDE-enabled datasets all have a CacheBlobs property that controls whether BLOB fields are cached locally by the BDE when an application reads BLOB records. By default, *CacheBlobs* is *True*, meaning that the BDE caches a local copy of BLOB fields. Caching BLOBs improves application performance by enabling the BDE to store local copies of BLOBs instead of fetching them repeatedly from the database server as a user scrolls through records.

In applications and environments where BLOBs are frequently updated or replaced, and a fresh view of BLOB data is more important than application performance, you can set *CacheBlobs* to *False* to ensure that your application always sees the latest version of a BLOB field.

See Also

Associating a Dataset with Database and Session Connections (see page 1668)

Working with BDE Handle Properties (see page 1671)

3.2.3.7.7.4 Controlling Read/Write Access to Local Tables

Like any table type dataset, *TTable* lets you control read and write access (see page 1587) by your application using the ReadOnly property.

In addition, for Paradox, dBASE, and FoxPro tables, *TTable* can let you control read and write access to tables by other applications. The Exclusive property controls whether your application gains sole read/write access to a Paradox, dBASE, or FoxPro table. To gain sole read/write access for these table types, set the table component's *Exclusive* property to *True* before opening the table. If you succeed in opening a table for exclusive access, other applications cannot read data from or write data to the table. Your request for exclusive access is not honored if the table is already in use when you attempt to open it.

The following statements open a table for exclusive access:

```
CustomersTable.Exclusive := True; {Set request for exclusive lock}
CustomersTable.Active := True; {Now open the table}
CustomersTable->Exclusive = true; // Set request for exclusive lock
CustomersTable->Active = true; // Now open the table
```

Note: You can attempt to set *Exclusive* on SQL tables, but some servers do not support exclusive table-level locking. Others may grant an exclusive lock, but permit other applications to read data from the table. For more information about exclusive locking of database tables on your server, see your server documentation.

See Also

Specifying a dBASE Index File (see page 1672)

Renaming Local Tables (see page 1671)

Specifying the Table Type for Local Tables (see page 1673)

3.2.3.7.7.5 Importing Data from Another Table

You can use a table component's BatchMove*method to import data from another table. BatchMove can*

- Copy records from another table into this table.
- Update records in this table that occur in another table.
- Append records from another table to the end of this table.
- Delete records in this table that occur in another table.

BatchMove takes two parameters: the name of the table from which to import data, and a mode specification that determines which import operation to perform. The following table describes the possible settings for the mode specification:

BatchMove import modes

Value	Meaning
batAppend	Append all records from the source table to the end of this table.
batAppendUpdate	Append all records from the source table to the end of this table and update existing records in this table with matching records from the source table.
batCopy	Copy all records from the source table into this table.
batDelete	Delete all records in this table that also appear in the source table.
batUpdate	Update existing records in this table with matching records from the source table.

For example, the following code updates all records in the current table with records from the *Customer* table that have the same values for fields in the current index:

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
Table1->BatchMove("CUSTOMER.DB", batUpdate);
```

BatchMove returns the number of records it imports successfully.

Warning: Importing records using the *batCopy* mode overwrites existing records. To preserve existing records use *batAppend* instead.

BatchMove performs only some of the batch operations supported by the BDE. Additional functions are available using the *TBatchMove* component. If you need to move a large amount of data between or among tables, use *TBatchMove* instead of calling a table's *BatchMove* method. For information about using *TBatchMove*, see Using TBatchMove (see page 1683)

See Also

Renaming Local Tables (see page 1671)

3.2.3.7.7.6 Working with BDE Handle Properties

You can use BDE-enabled datasets without ever needing to make direct API calls to the Borland Database Engine. The BDE-enabled datasets, in combination with database and session components, encapsulate much of the BDE functionality. However, if you need to make direct API calls to the BDE, you may need BDE handles for resources managed by the BDE. Many BDE APIs require these handles as parameters.

All BDE-enabled datasets include three read-only properties for accessing BDE handles at runtime:

- Handle is a handle to the BDE cursor that accesses the records in the dataset.
- DBHandle is a handle to the database that contains the underlying tables or stored procedure.
- DBLocale is a handle to the BDE language driver for the dataset. The locale controls the sort order and character set used for string data.

These properties are automatically assigned to a dataset when it is connected to a database server through the BDE.

See Also

Associating a Dataset with Database and Session Connections (see page 1668)

Caching BLOBs (see page 1670)

3.2.3.7.7.7 Renaming a Table

To rename a Paradox or dBASE table at runtime, call the table's RenameTable method. For example, the following statement renames the Customer table to CustInfo:

```
Customer.RenameTable('CustInfo');
Customer->RenameTable("CustInfo");
```

See Also

Controlling Read/Write Access to Local Tables (see page 1670)

Specifying a dBASE Index File (see page 1672)

Specifying the Table Type for Local Tables (see page 1673)

3.2.3.7.7.8 Specifying a dBASE Index File

For most servers, you use the methods common to all table type datasets to specify an index. These methods are described in Sorting records with indexes (see page 1609).

For dBASE tables that use non-production index files or dBASE III PLUS-style indexes (*.NDX), however, you must use the *IndexFiles and IndexNameproperties instead. Set the IndexFiles property to the name of the non-production index file or list the .NDX files. Then, specify one index in the IndexName property to have it actively sorting the dataset.*

*At design time, click the ellipsis button in the IndexFiles property value in the **Object Inspector** to invoke the Index Files editor. To add one non-production index file or .NDX file: click the Add button in the Index Files dialog and select the file from the Open dialog. Repeat this process once for each non-production index file or .NDX file. Click the OK button in the Index Files dialog after adding all desired indexes.*

This same operation can be performed programmatically at runtime. To do this, access the IndexFiles property using properties and methods of string lists. When adding a new set of indexes, first call the Clear method of the table's IndexFiles property to remove any existing entries. Call the Add method to add each non-production index file or .NDX file:

```
with Table2.IndexFiles do begin
  Clear;
  Add('Bystate.ndx');
  Add('Byzip.ndx');
  Add('Fullname.ndx');
  Add('St_name.ndx');
end;
Table2->IndexFiles->Clear();
Table2->IndexFiles->Add("Bystate.ndx");
Table2->IndexFiles->Add("Byzip.ndx");
Table2->IndexFiles->Add("Fullname.ndx");
Table2->IndexFiles->Add("St_name.ndx");
```

After adding any desired non-production or .NDX index files, the names of individual indexes in the index file are available, and can be assigned to the *IndexName* property. The index tags are also listed when using the *GetIndexNames* method and when inspecting index definitions through the *TIndexDef* objects in the *IndexDefs* property. Properly listed .NDX files are automatically updated as data is added, changed, or deleted in the table (regardless of whether a given index is used in the IndexName property).

In the example below, the *IndexFiles* for the *AnimalsTable* table component is set to the non-production index file ANIMALS.MDX, and then its *IndexName* property is set to the index tag called "NAME":

```
AnimalsTable.IndexFiles.Add('ANIMALS.MDX');
AnimalsTable.IndexName := 'NAME';
AnimalsTable->IndexFiles->Add("ANIMALS.MDX");
AnimalsTable->IndexName = "NAME";
```

Once you have specified the index file, using non-production or .NDX indexes works the same as any other index. Specifying an index name sorts the data in the table and makes it available for indexed-based searches, ranges, and (for non-production indexes) master-detail linking. See Using table type datasets (see page 1620) for details on these uses of indexes.

There are two special considerations when using dBASE III PLUS-style .NDX indexes with *TTable* components. The first is that .NDX files cannot be used as the basis for master-detail links. The second is that when activating a .NDX index with the *IndexName* property, you must include the .NDX extension in the property value as part of the index name:

```
with Table1 do begin
  IndexName := 'ByState.NDX';
  FindKey(['CA']);
end;
Table1->IndexName = "ByState.NDX";
TVarRec vr = ("NE");
Table1->FindKey(&vr, 0);
```

See Also

Controlling Read/Write Access to Local Tables (see page 1670)

Renaming Local Tables (see page 1671)

Specifying the Table Type for Local Tables (see page 1673)

3.2.3.7.7.9 Specifying the Table Type for Local Tables

If an application accesses Paradox, dBASE, FoxPro, or comma-delimited ASCII text tables, then the BDE uses the TableType property to determine the table's type (its expected structure). *TableType* is not used when *TTable* represents an SQL-based table on a database server.

By default *TableType* is set to *ttDefault*. When *TableType* is *ttDefault*, the BDE determines a table's type from its filename extension. The following table summarizes the file extensions recognized by the BDE and the assumptions it makes about a table's type:

Table types recognized by the BDE based on file extension

Extension	Table Type
No file extension	Paradox
.DB	Paradox
.DBF	dBASE
.TXT	ASCII text

If your local Paradox, dBASE, and ASCII text tables use the file extensions as described in the previous table, then you can leave *TableType* set to *ttDefault*. Otherwise, your application must set *TableType* to indicate the correct table type. The following table indicates the values you can assign to *TableType*:

TableType values

Value	Table Type
ttDefault	Table type determined automatically by the BDE
ttParadox	Paradox
ttDBase	dBASE
ttFoxPro	FoxPro
ttASCII	Comma-delimited ASCII text

See Also

Controlling Read/write Access to Local Tables (see page 1670)

Specifying a dBASE Index File (see page 1672)

Renaming Local Tables (see page 1671)

3.2.3.7.7.10 Using TQuery

TQuery represents a single Data Definition Language (DDL) or Data Manipulation Language (DML) statement (For example, a SELECT, INSERT, DELETE, UPDATE, CREATE INDEX, or ALTER TABLE command). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. *TQuery* implements all of the basic functionality introduced by TDataSet (see page 1632), as well as all of the special features typical of query-type datasets (see page 1618).

Because *TQuery* is a BDE-enabled dataset, it must usually be associated with a database and a session (see page 1668). (The one exception is when you use the *TQuery* for a heterogeneous query.) You specify the SQL statement for the query by setting the SQL property.

A TQuery component can access data in:

- Paradox or dBASE tables, using Local SQL, which is part of the BDE. Local SQL is a subset of the SQL-92 specification. Most DML is supported and enough DDL syntax to work with these types of tables. See the local SQL help, LOCALSQL.HLP, for details on supported SQL syntax.
- Local InterBase Server databases, using the InterBase engine. For information on InterBase's SQL-92 standard SQL syntax support and extended syntax support, see the InterBase *Language Reference.*
- Databases on remote database servers such as Oracle, Sybase, MS-SQL Server, Informix, DB2, and InterBase. You must install the appropriate SQL Link driver and client software (vendor-supplied) specific to the database server to access a remote server. Any standard SQL syntax supported by these servers is allowed. For information on SQL syntax, limitations, and extensions, see the documentation for your particular server.

The following topics discuss features that are unique to *TQuery* components (as opposed to other query-type datasets):

- Creating Heterogeneous Queries. (see page 1676)
- Obtaining an Editable Result Set (see page 1676)
- Updating Read-only Result Sets (see page 1677)

See Also

Using TTable (see page 70)

Using TStoredProc (see page 69)

3.2.3.7.7.11 Using TStoredProc

TStoredProc represents a stored procedure. It implements all of the basic functionality introduced by TDataSet (see page 1632), as well as most of the special features typical of stored procedure-type datasets (see page 1618).

Because *TStoredProc* is a BDE-enabled dataset, it must be associated with a database and a session (see page 1668). Once the dataset is associated with a database and session, you can bind it to a particular stored procedure by setting the StoredProcName property.

TStoredProc differs from other stored procedure-type datasets in the following ways:

- It gives you greater control over how to bind parameters (see page 1669).
- It provides support for Oracle overloaded stored procedures (see page 1677).

See Also

Using TTable (see page 70)

Using TQuery (see page 63)

3.2.3.7.7.12 Using TTable

TTable encapsulates the full structure of and data in an underlying database table. It implements all of the basic functionality

introduced by TDataSet (see page 1632), as well as all of the special features typical of table type datasets (see page 1620).

Because *TTable* is a BDE-enabled dataset, it must be associated with a database and a session (see page 1668). Once the dataset is associated with a database and session, you can bind it to a particular database table by setting the TableName property and, if you are using a Paradox, dBASE, FoxPro, or comma-delimited ASCII text table, the TableType property.

Note: The table must be closed when you change its association to a database, session, or database table, or when you set the *TableType* property. However, before you close the table to change these properties, first post or discard any pending changes. If cached updates are enabled, call the ApplyUpdates method to write the posted changes to the database.

TTable components are unique in the support they offer for local database tables (Paradox, dBASE, FoxPro, and comma-delimited ASCII text tables). The following topics describe the special properties and methods that implement this support:

- Specifying the Table Type for Local Tables (see page 1673)
- Controlling Read/Write Access to Local Tables (see page 1670)
- Specifying a dBASE Index File (see page 1672)
- Renaming Local Tables (see page 1671)

In addition, *TTable* components can take advantage of the BDE's support for batch operations (table level operations to append, update, delete, or copy entire groups of records). This support is described in Importing data from another table (see page 1670).

See Also

Using TQuery (see page 63)

Using TStoredProc (see page 69)

3.2.3.7.7.13 Using BDE-enabled Datasets

BDE-enabled datasets use the Borland Database Engine (BDE) to access data. They inherit the common dataset capabilities described in Understanding datasets (see page 1632), using the BDE to provide the implementation. In addition, all BDE datasets add properties, events, and methods for

- Associating a dataset with database and session connections (see page 1668).
- Caching BLOBs (see page 1670).
- Obtaining a BDE handle (see page 1671).

There are three BDE-enabled datasets:

- *TTable*, a table type dataset (see page 1620) that represents all of the rows and columns of a single database table. See Using TTable (see page 1674) for a description of features unique to *TTable*.
- *TQuery*, a query-type dataset (see page 1618) that encapsulates an SQL statement and enables applications to access the resulting records, if any. See Using TQuery (see page 1674) for a description of features unique to *TQuery*.
- *TStoredProc*, a stored procedure-type dataset (see page 1618) that executes a stored procedure that is defined on a database server. See Using TStoredProc (see page 1674) for a description of features unique to *TStoredProc*.

  Note: In addition to the three types of BDE-enabled datasets, there is a BDE-based client dataset (*TBDEClientDataSet*) that can be used for caching updates (see page 1731).

See Also

Connecting to Databases with TDatabase (see page 1646)

Managing Database Sessions Using TSession (see page 1664)

3.2.3.7.7.14 Creating Heterogenous Queries

TQuery supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table. When you execute a heterogeneous query, the BDE parses and processes the query using Local SQL. Because BDE uses Local SQL, extended, server-specific SQL syntax is not supported.

To perform a heterogeneous query

1. Define separate BDE aliases for each database accessed in the query using the BDE Administration tool or the SQL explorer.
2. Leave the *DatabaseName* property of the *TQuery* blank; the names of the databases used will be specified in the SQL statement.
3. In the SQL property, specify the SQL statement to execute. Precede each table name in the statement with the BDE alias for the table's database, enclosed in colons. This whole reference is then enclosed in quotation marks.
4. Set any parameters for the query in the *Params* property.
5. Call *Prepare* to prepare the query for execution prior to executing it for the first time.
6. Call *Open* or *ExecSQL* depending on the type of query you are executing.

For example, suppose you define an alias called Oracle1 for an Oracle database that has a CUSTOMER table, and Sybase1 for a Sybase database that has an ORDERS table. A simple query against these two tables would be:

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
  JOIN ":Sybase1:ORDERS"
    ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

As an alternative to using a BDE alias to specify the database in a heterogeneous query, you can use a *TDatabase* component. Configure the *TDatabase* as normal to point to the database, set the *TDatabase.DatabaseName* to an arbitrary but unique value, and then use that value in the SQL statement instead of a BDE alias name.

See Also

Obtaining an Editable Result Set (see page 1676)

Updating Read-only Result Sets (see page 1677)

3.2.3.7.7.15 Obtaining an Editable Result Set

To request a result set that users can edit in data-aware controls, set a query component's RequestLive property to True. Setting *RequestLive* to True does not guarantee a live result set, but the BDE attempts to honor the request whenever possible. There are some restrictions on live result set requests, depending on whether the query uses the local SQL parser or a server's SQL parser.

- Queries where table names are preceded by a BDE database alias (as in heterogeneous queries) and queries executed against Paradox or dBASE are parsed by the BDE using Local SQL. When queries use the local SQL parser, the BDE offers expanded support for updatable, live result sets in both single table and multi-table queries. When using Local SQL, a live result set for a query against a single table or view is returned if the query does not contain any of the following:
- DISTINCT in the SELECT clause
- Joins (inner, outer, or UNION)
- Aggregate functions with or without GROUP BY or HAVING clauses
- Base tables or views that are not updatable
- Subqueries
- ORDER BY clauses not based on an index
- Queries against a remote database server are parsed by the server. If the *RequestLive* property is set to True, the SQL

statement must abide by Local SQL standards in addition to any server-imposed restrictions because the BDE needs to use it for conveying data changes to the table. A live result set for a query against a single table or view is returned if the query does not contain any of the following:

- A DISTINCT clause in the SELECT statement
- Aggregate functions, with or without GROUP BY or HAVING clauses
- References to more than one base table or updatable views (joins)
- Subqueries that reference the table in the FROM clause or other tables

If an application requests and receives a live result set, the *CanModify* property of the query component is set to *True*. Even if the query returns a live result set, you may not be able to update the result set directly if it contains linked fields or you switch indexes before attempting an update. If these conditions exist, you should treat the result set as a read-only result set, and update it accordingly (see page 1677).

If an application requests a live result set, but the SELECT statement syntax does not allow it, the BDE returns either

- A read-only result set for queries made against Paradox or dBASE.
- An error code for SQL queries made against a remote server.

See Also

Updating Read-only Result Sets (see page 1677)

3.2.3.7.7.16 Updating a Read-only Result Set

Applications can update data returned in a read-only result set if they are using cached updates.

If you are using a client dataset to cache updates (see page 1731), the client dataset or its associated provider can automatically generate the SQL for applying updates unless the query represents multiple tables. If the query represents multiple tables, you must indicate how to apply the updates:

If all updates are applied to a single database table, you can indicate the underlying table to update in an OnGetTableName event handler.

If you need more control over applying updates, you can associate the query with an update object (TUpdateSQL). A provider automatically uses this update object to apply updates:

- Associate the update object with the query by setting the query's UpdateObject property to the *TUpdateSQL* object you are using.
- Set the update object's ModifySQL, InsertSQL, and DeleteSQL properties to SQL statements that perform the appropriate updates for your query's data.

You must use an update object if you are using the BDE to cache updates (see page 1689).

Note: For more information on using update objects, see Using update objects to update a dataset (see page 1692).

See Also

Obtaining an Editable Result Set (see page 1676)

Creating Heterogenous Queries (see page 1676)

3.2.3.7.7.17 Working with Oracle Overloaded Stored Procedures

Oracle servers allow overloading of stored procedures; overloaded procedures are different procedures with the same name. The stored procedure component's Overload property enables an application to specify the procedure to execute.

If *Overload* is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then the stored procedure component executes the first stored procedure it finds on the Oracle server that has the overloaded name; if it is two (2), it executes the second, and so on.

Note: Overloaded stored procedures may take different input and output parameters. See your Oracle server documentation for more information.

See Also

Binding Parameters (see page 1669)

3.2.3.7.8 Using TBatchMove

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Topics

Name	Description
Creating a Batch Move Component (see page 1679)	
Executing a Batch Move (see page 1680)	Use the Execute method to execute a previously prepared batch operation at runtime. For example, if *BatchMoveAdd* is the name of a batch move component, the following statement executes it:
The Data Dictionary (see page 1680)	When you use the BDE to access your data, your application has access to the Data Dictionary. The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data. For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the **Object Inspector** to set the currency fields in each dataset by hand, you can associate those fields... more (see page 1680)
Tools for Working with the BDE (see page 1681)	One advantage of using the BDE as a data access mechanism is the wealth of supporting utilities that ship with Delphi. These utilities include: • **SQL Explorer** and **Database Explorer**: Delphi ships with one of these two applications, depending on which version you have purchased. Both Explorers enable you to • Examine existing database tables and structures. The SQL Explorer lets you examine and query remote SQL databases. • Populate tables with data • Create extended field attribute sets in the Data Dictionary (see page 1680) or associate them with fields in your application. • Create and manage BDE aliases. SQL Explorer lets you do the following... more (see page 1681)
Specifying a Batch Move Mode (see page 1682)	The Mode property specifies the operation a batch move component performs: ***Batch move modes***

Using TBatchMove (see page 1683)	TBatchMove encapsulates Borland Database Engine (BDE) features that let you to duplicate a dataset, append records from one dataset to another, update records in one dataset with records from another dataset, and delete records from one dataset that match records in another dataset. *TBatchMove* is most often used to: • Download data from a server to a local data source for analysis or other operations. • Move a desktop database into tables on a remote server as part of an upsizing operation. A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the... more (see page 1683)
Handling Batch Move Errors (see page 1684)	There are two types of errors that can occur in a batch move operation: data type conversion errors and integrity violations. TBatchMove has a number of properties that report on and control error handling. The AbortOnProblem property specifies whether to abort the operation when a data type conversion error occurs. If *AbortOnProblem* is *True*, the batch move operation is canceled when an error occurs. If *False*, the operation continues. You can examine the table you specify in the *ProblemTableName* to determine which records caused problems. The AbortOnKeyViol property indicates whether to abort the operation when a Paradox key... more (see page 1684)
Mapping Data Types (see page 1684)	In *batAppend* mode, a batch move component creates the destination table based on the column data types of the source table. Columns and types are matched based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on. To override the default column mappings, use the Mappings property. *Mappings* is a list of column mappings (one per line). This listing can take one of two forms. To map a column in the source table to a column of the same name in... more (see page 1684)

3.2.3.7.8.1 Creating a Batch Move Component

To create a batch move component

1. Place a table or query component for the dataset from which you want to import records (called the *Source* dataset) on a form or in a data module.
2. Place the dataset to which to move records (called the *Destination* dataset) on the form or data module.
3. Place a*TBatchMove component from the BDE category of the **Tool palette** in the data module or form, and set its Nameproperty to a unique value appropriate to your application.*
4. Set the Source property of the batch move component to the name of the table from which to copy, append, or update records. You can select tables from the drop-down list of available dataset components.
5. Set the Destination property to the dataset to create, append to, or update. You can select a destination table from the drop-down list of available dataset components.

- If you are appending, updating, or deleting, *Destination* must represent an existing database table.
- If you are copying a table and *Destination* represents an existing table, executing the batch move overwrites all of the current data in the destination table.
- If you are creating an entirely new table by copying an existing table, the resulting table has the name specified in the *Name* property of the table component to which you are copying. The resulting table type will be of a structure appropriate to the server specified by the *DatabaseName* property.

6. Set the Mode property to indicate the type of operation to perform. Valid operations are *batAppend* (the default), *batUpdate*, *batAppendUpdate*, *batCopy*, and *batDelete*. For information about these modes, see Specifying a batch move mode (see page 1682).
7. Optionally set the *Transliterate property. If Transliterate is True (the default), character data is translated from the Source dataset's character set to the Destination dataset's character set as necessary.*

8. Optionally set column mappings using the Mappings property. You need not set this property if you want batch move to match columns based on their position in the source and destination tables. For more information about mapping columns, see Mapping data types (see page 1684).
9. Optionally specify the ChangedTableName, KeyViolTableName, and ProblemTableName properties. Batch move stores problem records it encounters during the batch operation in the table specified by *ProblemTableName*. If you are updating a Paradox table through a batch move, key violations can be reported in the table you specify in *KeyViolTableName*. *ChangedTableName* lists all records that changed in the destination table as a result of the batch move operation. If you do not specify these properties, these error tables are not created or used. For more information about handling batch move errors, see Handling batch move errors (see page 1684).

See Also

Executing a Batch Move (see page 1680)

3.2.3.7.8.2 Executing a Batch Move

Use the Execute method to execute a previously prepared batch operation at runtime. For example, if *BatchMoveAdd* is the name of a batch move component, the following statement executes it:

```
BatchMoveAdd.Execute;
BatchMoveAdd->Execute();
```

You can also execute a batch move at design time by right clicking the mouse on a batch move component and choosing Execute from the context menu.

The MovedCount property keeps track of the number of records that are moved when a batch move executes.

The RecordCount property specifies the maximum number of records to move. If *RecordCount* is zero, all records are moved, beginning with the first record in the source dataset. If *RecordCount* is a positive number, a maximum of *RecordCount* records are moved, beginning with the current record in the source dataset. If *RecordCount* is greater than the number of records between the current record in the source dataset and its last record, the batch move terminates when the end of the source dataset is reached. You can examine *MoveCount* to determine how many records were actually transferred.

See Also

Creating a Batch Move Component (see page 1679)

Specifying a Batch Move Mode (see page 1682)

Mapping Data Types (see page 1684)

Handling Batch Move Errors (see page 1684)

3.2.3.7.8.3 The Data Dictionary

When you use the BDE to access your data, your application has access to the Data Dictionary. The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the **Object Inspector** to set the currency fields in each dataset by hand, you can associate those fields with an extended field attribute set in the data dictionary. Using the data dictionary ensures a consistent data appearance within and across the applications you create.

In a client/server environment, the Data Dictionary can reside on a remote server for additional sharing of information.

To learn how to create extended field attribute sets from the Fields editor at design time, and how to associate them with fields throughout the datasets in your application, see Creating attribute sets for field components (see page 1870). To learn more about creating a data dictionary and extended field attributes with the SQL and Database Explorers, see their respective online

help files.

Note: A programming interface to the Data Dictionary is available in the drintf unit (located in the lib directory). This interface supplies the following methods:

Data Dictionary interface

Routine	Use
DictionaryActive	Indicates if the data dictionary is active.
DictionaryDeactivate	Deactivates the data dictionary.
IsNullID	Indicates whether a given ID is a null ID
FindDatabaseID	Returns the ID for a database given its alias.
FindTableID	Returns the ID for a table in a specified database.
FindFieldID	Returns the ID for a field in a specified table.
FindAttrID	Returns the ID for a named attribute set.
GetAttrName	Returns the name an attribute set given its ID.
GetAttrNames	Executes a callback for each attribute set in the dictionary.
GetAttrID	Returns the ID of the attribute set for a specified field.
NewAttr	Creates a new attribute set from a field component.
UpdateAttr	Updates an attribute set to match the properties of a field.
CreateField	Creates a field component based on stored attributes.
UpdateField	Changes the properties of a field to match a specified attribute set.
AssociateAttr	Associates an attribute set with a given field ID.
UnassociateAttr	Removes an attribute set association for a field ID.
GetControlClass	Returns the control class for a specified attribute ID.
QualifyTableName	Returns a fully qualified table name (qualified by user name).
QualifyTableNameByName	Returns a fully qualified table name (qualified by user name).
HasConstraints	Indicates whether the dataset has constraints in the dictionary.
UpdateConstraints	Updates the imported constraints of a dataset.
UpdateDataset	Updates a dataset to the current settings and constraints in the dictionary.

See Also

BDE-based Architecture (see page 1638)

Tools for Working with the BDE (see page 1681)

Using Transactions with the BDE (see page 1699)

3.2.3.7.8.4 Tools for Working with the BDE

One advantage of using the BDE as a data access mechanism is the wealth of supporting utilities that ship with Delphi. These utilities include:

- **SQL Explorer** and **Database Explorer**: Delphi ships with one of these two applications, depending on which version you have purchased. Both Explorers enable you to

- Examine existing database tables and structures. The SQL Explorer lets you examine and query remote SQL databases.
- Populate tables with data
- Create extended field attribute sets in the Data Dictionary (see page 1680) or associate them with fields in your application.
- Create and manage BDE aliases.

SQL Explorer lets you do the following as well:

- Create SQL objects such as stored procedures on remote database servers.
- View the reconstructed text of SQL objects on remote database servers.
- Run SQL scripts.
- **SQL Monitor**: SQL Monitor lets you watch all of the communication that passes between the remote database server and the BDE. You can filter the messages you want to watch, limiting them to only the categories of interest. SQL Monitor is most useful when debugging your application.
- **Database Desktop**: If you are using Paradox or dBASE tables, Database Desktop lets you view and edit their data, create new tables, and restructure existing tables. Using Database Desktop affords you more control than using the methods of a *TTable* component (for example, it allows you to specify validity checks and language drivers). It provides the only mechanism for restructuring Paradox and dBASE tables other than making direct calls the BDE's API.

See Also

BDE-based Architecture (see page 1638)

Using TBatchMove (see page 1683)

The Data Dictionary (see page 1680)

Using Transactions with the BDE (see page 1699)

3.2.3.7.8.5 Specifying a Batch Move Mode

The Mode property specifies the operation a batch move component performs:

Batch move modes

Property	Purpose
batAppend	Append records to the destination table.
batUpdate	Update records in the destination table with matching records from the source table. Updating is based on the current index of the destination table.
batAppendUpdate	If a matching record exists in the destination table, update it. Otherwise, append records to the destination table.
batCopy	Create the destination table based on the structure of the source table. If the destination table already exists, it is dropped and recreated.
batDelete	Delete records in the destination table that match records in the source table.

Appending records

To append data, the destination dataset must represent an existing table. During the append operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary. If a conversion is not possible, an exception is thrown and the data is not appended.

Updating records

To update data, the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. During the update operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary.

Appending and updating records

To append and update data the destination dataset must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. Otherwise, data from the source dataset is appended to the destination dataset. During append and update operations, the BDE converts data to appropriate data types and sizes for the destination dataset, if necessary.

Copying datasets

To copy a source dataset, the destination dataset should not represent an exist table. If it does, the batch move operation overwrites the existing table with a copy of the source dataset.

If the source and destination datasets are maintained by different types of database engines, for example, Paradox and InterBase, the BDE creates a destination dataset with a structure as close as possible to that of the source dataset and automatically performs data type and size conversions as necessary.

Note: *TBatchMove* does not copy metadata structures such as indexes, constraints, and stored procedures. You must recreate these metadata objects on your database server as appropriate.

Deleting records

To delete data in the destination dataset, it must represent an existing table and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are deleted in the destination table.

See Also

Creating a Batch Move Component (see page 1679)

Mapping Data Types (see page 1684)

Executing a Batch Move (see page 1680)

Handling Batch Move Errors (see page 1684)

3.2.3.7.8.6 Using TBatchMove

TBatchMove encapsulates Borland Database Engine (BDE) features that let you to duplicate a dataset, append records from one dataset to another, update records in one dataset with records from another dataset, and delete records from one dataset that match records in another dataset. *TBatchMove* is most often used to:

- Download data from a server to a local data source for analysis or other operations.
- Move a desktop database into tables on a remote server as part of an upsizing operation.

A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate.

The following topics describe how to work with a TBatchMove component:

- Creating a Batch Move Component (see page 1679)
- Specifying a Batch Move Mode (see page 1682)
- Mapping Data Types (see page 1684)
- Executing a Batch Move (see page 1680)
- Handling Batch Move Errors (see page 1684)

See Also

BDE-based Architecture (see page 1638)

Tools for Working with the BDE (see page 1681)

Using Transactions with the BDE (see page 1699)

The Data Dictionary (see page 1680)

3.2.3.7.8.7 Handling Batch Move Errors

There are two types of errors that can occur in a batch move operation: data type conversion errors and integrity violations. TBatchMove has a number of properties that report on and control error handling.

The AbortOnProblem property specifies whether to abort the operation when a data type conversion error occurs. If *AbortOnProblem* is *True*, the batch move operation is canceled when an error occurs. If *False*, the operation continues. You can examine the table you specify in the *ProblemTableName* to determine which records caused problems.

The AbortOnKeyViol property indicates whether to abort the operation when a Paradox key violation occurs.

The *ProblemCount property indicates the number of records that could not be handled in the destination table without a loss of data. If AbortOnProblem is True, this number is one, since the operation is aborted when an error occurs.*

The following properties enable a batch move component to create additional tables that document the batch move operation:

- ChangedTableName, if specified, creates a local Paradox table containing all records in the destination table that changed as a result of an update or delete operation.
- KeyViolTableName, if specified, creates a local Paradox table containing all records from the source table that caused a key violation when working with a Paradox table. If *AbortOnKeyViol* is *True*, this table will contain at most one entry since the operation is aborted on the first problem encountered.
- ProblemTableName, if specified, creates a local Paradox table containing all records that could not be posted in the destination table due to data type conversion errors. For example, the table could contain records from the source table whose data had to be trimmed to fit in the destination table. If *AbortOnProblem* is *True*, there is at most one record in this table since the operation is aborted on the first problem encountered.

  Note: If *ProblemTableName* is not specified, the data in the record is trimmed and placed in the destination table.

See Also

Creating a Batch Move Component (see page 1679)

Specifying a Batch Move Mode (see page 1682)

Mapping Data Types (see page 1684)

Executing a Batch Move (see page 1680)

3.2.3.7.8.8 Mapping Data Types

In *batAppend* mode, a batch move component creates the destination table based on the column data types of the source table. Columns and types are matched based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on.

To override the default column mappings, use the Mappings property. *Mappings* is a list of column mappings (one per line). This listing can take one of two forms. To map a column in the source table to a column of the same name in the destination table, you can use a simple listing that specifies the column name to match. For example, the following mapping specifies that a column named *ColName* in the source table should be mapped to a column of the same name in the destination table:

```
ColName
```

To map a column named *SourceColName* in the source table to a column named *DestColName* in the destination table, the syntax is as follows:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, a batch move operation attempts a "best fit". It trims character

data types, if necessary, and attempts to perform a limited amount of conversion, if possible. For example, mapping a CHAR(10) column to a CHAR(5) column will result in trimming the last five characters from the source column.

As an example of conversion, if a source column of character data type is mapped to a destination of integer type, the batch move operation converts a character value of '5' to the corresponding integer value. Values that cannot be converted generate errors. For more information about errors, see Handling batch move errors (see page 1684).

When moving data between different table types, a batch move component translates data types as appropriate based on the dataset's server types. See the BDE online help file for the latest tables of mappings among server types.

Note: To batch move data to an SQL server database, you must have that database server and a version of Delphi with the appropriate SQL Link installed, or you can use ODBC if you have the proper third party ODBC drivers installed.

See Also

Creating a Batch Move Component (see page 1679)

Specifying a Batch Move Mode (see page 1682)

Executing a Batch Move (see page 1680)

Handling Batch Move Errors (see page 1684)

3.2.3.7.9 Using the BDE to cache updates

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Topics

Name	Description
Applying Cached Updates Using a Database (see page 1687)	To apply cached updates to one or more datasets in the context of a database connection, call the database component's ApplyUpdates method. The following code applies updates to the *CustomersQuery* dataset in response to a button click event:
Composing Update SQL Statements (see page 1687)	At design time, you can use the Update SQL editor to write the SQL statements for the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. If you do not use the Update SQL editor, or if you want to modify the generated statements, you should keep in mind the following guidelines when writing statements to delete, insert, and modify records in the base table. The *DeleteSQL* property should contain only an SQL statement with the DELETE command. The base table to be updated must be named in the FROM clause. So that the SQL statement only deletes the record in the... more (see page 1687)
Enabling BDE-based Cached Updates (see page 1688)	To use the BDE for cached updates, the BDE-enabled dataset must indicate that it should cache updates. This is specified by setting the CachedUpdates property to *True*. When you enable cached updates, a copy of all records is cached in local memory. Users view and edit this local copy of data. Changes, insertions, and deletions are also cached in memory. They accumulate in memory until the application applies those changes to the database server. If changed records are successfully applied to the database, the record of those changes are freed in the cache. The dataset caches all updates until... more (see page 1688)
Understanding Parameter Substitution in Update SQL Statements (see page 1689)	Update SQL statements use a special form of parameter substitution that enables you to substitute old or new field values in record updates. When the Update SQL editor generates its statements, it determines which field values to use. When you write the update SQL, you specify the field values to use. When the parameter name matches a column name in the table, the new value in the field in the cached update for the record is automatically used as the value for the parameter. When the parameter name matches a column name prefixed by the string "OLD_", then the old... more (see page 1689)

Using the BDE to Cache Updates (see page 1689)	The recommended approach for caching updates is to use a client dataset (*TBDEClientDataSet*) or to connect the BDE-dataset to a client dataset using a dataset provider. The advantages of using a client dataset are discussed in Using a client dataset to cache updates (see page 1731). For simple cases, however, you may choose to use the BDE to cache updates instead. BDE-enabled datasets and *TDatabase* components provide built-in properties, methods, and events for handling cached updates. Most of these correspond directly to the properties, methods, and events that you use with client datasets and dataset providers when using a client... more (see page 1689)
Using the Update SQL Editor (see page 1691)	
Applying BDE-based Cached Updates (see page 1691)	Applying updates is a two-phase process that should occur in the context of a database component's transaction so that your application can recover gracefully from errors. For information about transaction handling with database components, see Managing Transactions (see page 1498). When applying updates under database transaction control, the following events take place: 1. A database transaction starts. 2. Cached updates are written to the database (phase 1). If you provide it, an *OnUpdateRecord* event is triggered once for each record written to the database. If an error occurs when a record is applied to the database, the *OnUpdateError* event is triggered if you provide... more (see page 1691)
Using Update Objects to Update a Dataset (see page 1692)	When the BDE-enabled dataset represents a stored procedure or a query that is not "live", it is not possible to apply updates directly from the dataset. Such datasets may also cause a problem when you use a client dataset to cache updates. Whether you are using the BDE or a client dataset to cache updates, you can handle these problem datasets by using an update object.
Creating an OnUpdateRecord Event Handler (see page 1693)	When a BDE-enabled dataset applies its cached updates, it iterates through the changes recorded in its cache, attempting to apply them to the corresponding records in the base table. As the update for each changed, deleted, or newly inserted record is about to be applied, the dataset component's *OnUpdateRecord* event fires. Providing a handler for the *OnUpdateRecord* event allows you to perform actions just before the current record's update is actually applied. Such actions can include special data validation, updating other tables, special parameter substitution, or executing multiple update objects (see page 1692). A handler for the *OnUpdateRecord* event affords you greater... more (see page 1693)
Applying Cached Updates with Dataset Component Methods (see page 1695)	You can apply updates for individual BDE-enabled datasets directly using the dataset's ApplyUpdates and CommitUpdates methods. Each of these methods encapsulate one phase of the update process: 1. *ApplyUpdates* writes cached changes to a database (phase 1). 2. *CommitUpdates* clears the internal cache when the database write is successful (phase 2). The following code illustrates how you apply updates within a transaction for the *CustomerQuery* dataset:
Creating SQL Statements for Update Components (see page 1696)	To update a record in an associated dataset, an update object uses one of three SQL statements. Each update object can only update a single table, so the object's update statements must each reference the same base table. The three SQL statements delete, insert, and modify records cached for update. You must provide these statements as update object's *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. You can provide these values at design time or at runtime. For example, the following code specifies a value for the *DeleteSQL* property at runtime:
Handling Cached Update Errors (see page 1696)	The Borland Database Engine (BDE) specifically checks for user update conflicts and other conditions when attempting to apply updates, and reports any errors. The dataset component's OnUpdateError event enables you to catch and respond to errors. You should create a handler for this event if you use cached updates. If you do not, and an error occurs, the entire update operation fails. Here is the skeleton code for an *OnUpdateError* event handler:

3.2.3.7.9.1 Applying Cached Updates Using a Database

To apply cached updates to one or more datasets in the context of a database connection, call the database component's ApplyUpdates method. The following code applies updates to the *CustomersQuery* dataset in response to a button click event:

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  // for local databases such as Paradox, dBASE, and FoxPro
  // set TransIsolation to DirtyRead
  if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
  Database1.ApplyUpdates([CustomersQuery]);
end;
void __fastcall TForm1::ApplyButtonClick(TObject *Sender)
{
  // for local databases such as Paradox, dBASE, and FoxPro
  // set TransIsolation to DirtyRead
  if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
    Database1->TransIsolation = tiDirtyRead;
  Database1->ApplyUpdates(&CustomersQuery,0);
}
```

The above sequence writes cached updates to the database in the context of an automatically-generated transaction. If successful, it commits the transaction and then commits the cached updates. If unsuccessful, it rolls back the transaction and leaves the update cache unchanged. In this latter case, you should handle cached update errors through a dataset's *OnUpdateError* event. For more information about handling update errors, see Handling cached update errors. (see page 1696)

The main advantage to calling a database component's *ApplyUpdates* method is that you can update any number of dataset components that are associated with the database. The parameter for the *ApplyUpdates* method for a database is an array of *TDBDataSet*. For example, the following code applies updates for two queries:

```
if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
  Database1.TransIsolation := tiDirtyRead;
Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);
TDBDataSet* ds[] = {CustomerQuery, OrdersQuery};
if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
  Database1->TransIsolation = tiDirtyRead;
Database1->ApplyUpdates(ds,1);
```

See Also

Applying Cached Updates with Dataset Component Methods (see page 1695)

Handling Cached Update Errors (see page 1696)

3.2.3.7.9.2 Composing Update SQL Statements

At design time, you can use the Update SQL editor to write the SQL statements for the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. If you do not use the Update SQL editor, or if you want to modify the generated statements, you should keep in mind the following guidelines when writing statements to delete, insert, and modify records in the base table.

The *DeleteSQL* property should contain only an SQL statement with the DELETE command. The base table to be updated must be named in the FROM clause. So that the SQL statement only deletes the record in the base table that corresponds to the record deleted in the update cache, use a WHERE clause. In the WHERE clause, use a parameter for one or more fields to uniquely identify the record in the base table that corresponds to the cached update record. If the parameters are named the same as the field and prefixed with "OLD_", the parameters are automatically given the values from the corresponding field from the cached update record. If the parameter are named in any other manner, you must supply the parameter values.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

Some table types might not be able to find the record in the base table when fields used to identify the record contain NULL values. In these cases, the delete update fails for those records. To accommodate this, add a condition for those fields that might contain NULLs using the IS NULL predicate (in addition to a condition for a non-NULL value). For example, when a FirstName field may contain a NULL value:

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
  ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

The *InsertSQL* statement should contain only an SQL statement with the INSERT command. The base table to be updated must be named in the INTO clause. In the VALUES clause, supply a comma-separated list of parameters. If the parameters are named the same as the field, the parameters are automatically given the value from the cached update record. If the parameter are named in any other manner, you must supply the parameter values. The list of parameters supplies the values for fields in the newly inserted record. There must be as many value parameters as there are fields listed in the statement.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

The *ModifySQL* statement should contain only an SQL statement with the UPDATE command. The base table to be updated must be named in the FROM clause. Include one or more value assignments in the SET clause. If values in the SET clause assignments are parameters named the same as fields, the parameters are automatically given values from the fields of the same name in the updated record in the cache. You can assign additional field values using other parameters, as long as the parameters are not named the same as any fields and you manually supply the values. As with the *DeleteSQL* statement, supply a WHERE clause to uniquely identify the record in the base table to be updated using parameters named the same as the fields and prefixed with "OLD_". In the update statement below, the parameter :ItemNo is automatically given a value and :Price is not.

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

Considering the above update SQL, take an example case where the application end-user modifies an existing record. The original value for the ItemNo field is 999. In a grid connected to the cached dataset, the end-user changes the ItemNo field value to 123 and Amount to 20. When the ApplyUpdates method is invoked, this SQL statement affects all records in the base table where the ItemNo field is 999, using the old field value in the parameter :OLD_ItemNo. In those records, it changes the ItemNo field value to 123 (using the parameter :ItemNo, the value coming from the grid) and Amount to 20.

See Also

Understanding Parameter Substitution in Update SQL Statements (see page 1689)

Creating SQL Statements at Design Time (see page 1691)

3.2.3.7.9.3 Enabling BDE-based Cached Updates

To use the BDE for cached updates, the BDE-enabled dataset must indicate that it should cache updates. This is specified by setting the CachedUpdates property to *True*. When you enable cached updates, a copy of all records is cached in local memory. Users view and edit this local copy of data. Changes, insertions, and deletions are also cached in memory. They accumulate in memory until the application applies those changes to the database server. If changed records are successfully applied to the database, the record of those changes are freed in the cache.

The dataset caches all updates until you set *CachedUpdates* to *False*. Applying cached updates does not disable further cached updates; it only writes the current set of changes to the database and clears them from memory. Canceling the updates by calling CancelUpdates removes all the changes currently in the cache, but does not stop the dataset from caching any subsequent changes.

Note: If you disable cached updates by setting *CachedUpdates* to *False*, any pending changes that you have not yet applied are discarded without notification. To prevent losing changes, test the UpdatesPending property before disabling cached updates.

See Also

Using the BDE to Cache Updates (see page 1689)

Using Update Objects to Update a Dataset (see page 1692)

Applying BDE Based Cached Updates (see page 1691)

3.2.3.7.9.4 Understanding Parameter Substitution in Update SQL Statements

Update SQL statements use a special form of parameter substitution that enables you to substitute old or new field values in record updates. When the Update SQL editor generates its statements, it determines which field values to use. When you write the update SQL, you specify the field values to use.

When the parameter name matches a column name in the table, the new value in the field in the cached update for the record is automatically used as the value for the parameter. When the parameter name matches a column name prefixed by the string "OLD_", then the old value for the field will be used. For example, in the update SQL statement below, the parameter :LastName is automatically filled with the new field value in the cached update for the inserted record.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

New field values are typically used in the *InsertSQL* and *ModifySQL* statements. In an update for a modified record, the new field value from the update cache is used by the UPDATE statement to replace the old field value in the base table updated.

In the case of a deleted record, there are no new values, so the *DeleteSQL* property uses the ":OLD_FieldName" syntax. Old field values are also normally used in the WHERE clause of the SQL statement for a modified or deletion update to determine which record to update or delete.

In the WHERE clause of an UPDATE or DELETE update SQL statement, supply at least the minimal number of parameters to uniquely identify the record in the base table that is updated with the cached data. For instance, in a list of customers, using just a customer's last name may not be sufficient to uniquely identify the correct record in the base table; there may be a number of records with "Smith" as the last name. But by using parameters for last name, first name, and phone number could be a distinctive enough combination. Even better would be a unique field value like a customer number.

Note: If you create SQL statements that contain parameters that do not refer the edited or original field values, the update object does not know how to bind their values. You can, however, do this manually, using the update object's Query property (see page 1640).

See Also

Composing Update SQL Statements (see page 1687)

Creating SQL Statements at Design Time (see page 1691)

3.2.3.7.9.5 Using the BDE to Cache Updates

The recommended approach for caching updates is to use a client dataset (*TBDEClientDataSet*) or to connect the BDE-dataset to a client dataset using a dataset provider. The advantages of using a client dataset are discussed in Using a client dataset to cache updates (see page 1731).

For simple cases, however, you may choose to use the BDE to cache updates instead. BDE-enabled datasets and *TDatabase* components provide built-in properties, methods, and events for handling cached updates. Most of these correspond directly to the properties, methods, and events that you use with client datasets and dataset providers when using a client dataset to cache updates. The following table lists these properties, events, and methods and the corresponding properties, methods and events on *TBDEClientDataSet*:

Properties, methods, and events for cached updates

On BDE-enabled Datasets (or TDatabase)	On TBDEClientDataSet	Purpose
CachedUpdates	Not needed for client datasets, which always cache updates.	Determines whether cached updates are in effect for the dataset.
UpdateObject	Use a *BeforeUpdateRecord* event handler, or, if using *TClientDataSet*, use the *UpdateObject* property on the BDE-enabled source dataset.	Specifies the update object for updating read-only datasets.
UpdatesPending	ChangeCount	Indicates whether the local cache contains updated records that need to be applied to the database.
UpdateRecordTypes	StatusFilter	Indicates the kind of updated records to make visible when applying cached updates.
UpdateStatus	UpdateStatus	Indicates if a record is unchanged, modified, inserted, or deleted.
OnUpdateError	OnReconcileError	An event for handling update errors on a record-by-record basis.
OnUpdateRecord	BeforeUpdateRecord	An event for processing updates on a record-by-record basis.
ApplyUpdates (database)	ApplyUpdates	Applies records in the local cache to the database.
CancelUpdates	CancelUpdates	Removes all pending updates from the local cache without applying them.
CommitUpdates	Reconcile	Clears the update cache following successful application of updates.
FetchAll	GetNextPacket (and PacketRecords)	Copies database records to the local cache for editing and updating.
RevertRecord	RevertRecord	Undoes updates to the current record if updates are not yet applied.

For an overview of the cached update process, see Overview of using cached updates (see page 1723).

The following topics describe in more detail on how to use the BDE to cache updates:

- Enabling BDE-based Cached Updates (see page 1688).
- Applying BDE-based Cached Updates (see page 1691).
- Using Update Objects to Update a Dataset (see page 1692).

  Note: Even if you are using a client dataset to cache updates, you may want to read the section about update objects. You can use update objects in the *BeforeUpdateRecord* event handler of *TBDEClientDataSet* or *TDataSetProvider* to apply updates from stored procedures or multi-table queries.

See Also

BDE-based Architecture (see page 1638)

Using TBatchMove (see page 1683)

Tools for Working with the BDE (see page 1681)

Using Transactions with the BDE (see page 1699)

The Data Dictionary (see page 1680)

3.2.3.7.9.6 Using the Update SQL Editor

To create the SQL statements for an update component

1. Using the **Object Inspector**, select the name of the update object from the drop-down list for the dataset's *UpdateObject* property. This step ensures that the Update SQL editor you invoke in the next step can determine suitable default values to use for SQL generation options.
2. Right-click the update object and select UpdateSQL Editor from the context menu. This displays the Update SQL editor. The editor creates SQL statements for the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties based on the underlying data set and on the values you supply to it.

The Update SQL editor has two pages. The Options page is visible when you first invoke the editor. Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.

The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired.

The Key Fields list box is used to specify the columns to use as keys during the update. For Paradox, dBASE, and FoxPro the columns you specify here must correspond to an existing index, but this is not a requirement for remote SQL databases. Instead of setting Key Fields you can click the Primary Keys button to choose key fields for the update based on the table's primary index. Click Dataset Defaults to return the selection lists to the original state: all fields selected as keys and all selected for update.

Check the Quote Field Names check box if your server requires quotation marks around field names.

After you specify a table, select key columns, and select update columns, click Generate SQL to generate the preliminary SQL statements to associate with the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. In most cases you will want or need to fine tune the automatically generated SQL statements.

To view and modify the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the *ModifySQL* property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

Warning: Keep in mind that generated SQL statements are starting points for creating update statements. You may need to modify these statements to make them execute correctly. For example, when working with data that contains NULL values, you need to modify the WHERE clause to read

```
WHERE field IS NULL
```

rather then using the generated field variable. Test each of the statements directly yourself before accepting them.

Use the Statement Type radio buttons to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

See Also

Understanding Parameter Substitution in Update SQL Statements (see page 1689)

Composing Update SQL Statements (see page 1687)

3.2.3.7.9.7 Applying BDE-based Cached Updates

Applying updates is a two-phase process that should occur in the context of a database component's transaction so that your application can recover gracefully from errors. For information about transaction handling with database components, see Managing Transactions (see page 1498).

When applying updates under database transaction control, the following events take place:

1. A database transaction starts.
2. Cached updates are written to the database (phase 1). If you provide it, an *OnUpdateRecord* event is triggered once for each

record written to the database. If an error occurs when a record is applied to the database, the *OnUpdateError* event is triggered if you provide one.

3. The transaction is committed if writes are successful or rolled back if they are not,

Write Status	Transaction
Successful	Database changes are committed, ending the database transaction. Cached updates are committed, clearing the internal cache buffer (phase 2).
Unsuccessful	Database changes are rolled back, ending the database transaction. Cached updates are not committed, remaining intact in the internal cache.

For information about creating and using an *OnUpdateRecord* event handler, see Creating an OnUpdateRecord Event Handler. (see page 1693) For information about handling update errors that occur when applying cached updates, see Handling Cached Update errors (see page 1696).

Note: Applying cached updates is particularly tricky when you are working with multiple datasets linked in a master/detail relationship because the order in which you apply updates to each dataset is significant. Usually, you must update master tables before detail tables, except when handling deleted records, where this order must be reversed. Because of this difficulty, it is strongly recommended that you use client datasets when caching updates in a master/detail form. Client datasets automatically handle all ordering issues with master/detail relationships.

There are two ways to apply BDE-based updates:

- You can apply updates using a database component (see page 1687) by calling its ApplyUpdates method. This method is the simplest approach, because the database handles all details of managing a transaction for the update process and of clearing the dataset's cache when updating is complete.
- You can apply updates for a single dataset (see page 1695) by calling the dataset's ApplyUpdates and CommitUpdates methods. When applying updates at the dataset level you must explicitly code the transaction that wraps the update process as well as explicitly call *CommitUpdates* to commit updates from the cache.

  Warning: To apply updates from a stored procedure or an SQL query that does not return a live result set, you must use *TUpdateSQL* to specify how to perform updates. For updates to joins (queries involving two or more tables), you must provide one *TUpdateSQL* object for each table involved, and you must use the *OnUpdateRecord* event handler to invoke these objects to perform the updates. See Using update objects to update a dataset (see page 1692) for details.

See Also

Using the BDE to Cache Updates (see page 1689)

Using Update Objects to Update a Dataset (see page 1692)

Enabling BDE-based Cached Updates (see page 1688)

3.2.3.7.9.8 Using Update Objects to Update a Dataset

When the BDE-enabled dataset represents a stored procedure or a query that is not "live", it is not possible to apply updates directly from the dataset. Such datasets may also cause a problem when you use a client dataset to cache updates. Whether you are using the BDE or a client dataset to cache updates, you can handle these problem datasets by using an update object.

To update a dataset

1. If you are using a client dataset, use an external provider component with *TClientDataSet* rather than *TBDEClientDataSet*. This is so you can set the *UpdateObject* property of the BDE-enabled source dataset (step 3).
2. Add a TUpdateSQL component to the same data module as the BDE-enabled dataset.
3. Set the BDE-enabled dataset component's UpdateObject property to the *TUpdateSQL* component in the data module.

4. Specify the SQL statements (see page 1696) needed to perform updates using the update object's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. You can use the Update SQL editor to help you compose these statements.
5. Close the dataset.
6. Set the dataset component's *CachedUpdates* property to *True* or link the dataset to the client dataset using a dataset provider.
7. Reopen the dataset.

Note: Sometimes, you need to use multiple update objects. For example, when updating a multi-table join or a stored procedure that represents data from multiple datasets, you must provide one *TUpdateSQL* object for each table you want to update. When using multiple update objects, you can't simply associate the update object with the dataset by setting the *UpdateObject* property. Instead, you must manually call the update object from an OnUpdateRecord event handler (when using the BDE to cache updates) or a BeforeUpdateRecord event handler (when using a client dataset).

The update object actually encapsulates three *TQuery* components. Each of these query components perform a single update task. One query component provides an SQL UPDATE statement for modifying existing records; a second query component provides an INSERT statement to add new records to a table; and a third component provides a DELETE statement to remove records from a table.

When you place an update component in a data module, you do not see the query components it encapsulates. They are created by the update component at runtime based on three update properties for which you supply SQL statements:

- ModifySQL specifies the UPDATE statement.
- InsertSQL specifies the INSERT statement.
- DeleteSQL specifies the DELETE statement.

At runtime, when the update component is used to apply updates, it:

1. Selects an SQL statement to execute based on whether the current record is modified, inserted, or deleted.
2. Provides parameter values to the SQL statement.
3. Prepares and executes the SQL statement to perform the specified update.

See Also

Using the BDE to Cache Updates (see page 1689)

Indicating What Records Are Modified (see page 1717)

Overview of BDE Based Cached Updates (see page 1688)

3.2.3.7.9.9 Creating an OnUpdateRecord Event Handler

When a BDE-enabled dataset applies its cached updates, it iterates through the changes recorded in its cache, attempting to apply them to the corresponding records in the base table. As the update for each changed, deleted, or newly inserted record is about to be applied, the dataset component's *OnUpdateRecord* event fires.

Providing a handler for the *OnUpdateRecord* event allows you to perform actions just before the current record's update is actually applied. Such actions can include special data validation, updating other tables, special parameter substitution, or executing multiple update objects (see page 1692). A handler for the *OnUpdateRecord* event affords you greater control over the update process.

Here is the skeleton code for an *OnUpdateRecord* event handler:

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
 UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { perform updates here... }
end;
void __fastcall TForm1::DataSetUpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  // Perform updates here...
```

```
}
```

The *DataSet* parameter specifies the cached dataset with updates.

The *UpdateKind* parameter indicates the type of update that needs to be performed for the current record. Values for *UpdateKind* are *ukModify*, *ukInsert*, and *ukDelete*. If you are using an update object, you need to pass this parameter to the update object when applying the update. You may also need to inspect this parameter if your handler performs any special processing based on the kind of update.

The *UpdateAction* parameter indicates whether you applied the update. Values for *UpdateAction* are *uaFail* (the default), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*. If your event handler successfully applies the update, change this parameter to *uaApplied* before exiting. If you decide not to update the current record, change the value to *uaSkip* to preserve unapplied changes in the cache. If you do not change the value for *UpdateAction*, the entire update operation for the dataset is aborted and an exception is raised. You can suppress the error message (raising a silent exception) by changing *UpdateAction* to *uaAbort*.

In addition to these parameters, you will typically want to make use of the *OldValue* and *NewValue* properties for the field component associated with the current record. *OldValue* gives the original field value that was fetched from the database. It can be useful in locating the database record to update. *NewValue* is the edited value in the update you are trying to apply.

Warning: An *OnUpdateRecord* event handler, like an *OnUpdateError* or *OnCalcFields* event handler, should never call any methods that change the current record in a dataset.

The following example illustrates how to use these parameters and properties. It uses a *TTable* component named *UpdateTable* to apply updates. In practice, it is easier to use an update object, but using a table illustrates the possibilities more clearly.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue, DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField', VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            UpdateTable.Edit;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukInsert:
          begin
            UpdateTable.Insert;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            UpdateTable.Post;
          end;
        ukDelete: UpdateTable.Delete;
      end;
        UpdateAction := uaApplied;
end;
void __fastcall TForm1::EmpAuditUpdateRecord(TDataSet *DataSet,
  TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  if (UpdateKind == ukInsert)
  {
    TVarRec values[2];
    for (int i = 0; i < 2; i++)
      values[i] = DataSet->Fields->Fields[i]->NewValue;
    UpdateTable->AppendRecord(values, 1);
  }
  else
  {
    TLocateOptions lo;
    lo.Clear();
    if (UpdateTable->Locate("KeyField", DataSet->Fields->Fields[0]->OldValue, lo))
```

```
        switch (UpdateKind)
        {
          case ukModify:
            UpdateTable->Edit();
            UpdateTable->Fields->Fields[1]->Value = DataSet->Fields->Fields[1]->Value;
            UpdateTable->Post();
            break;
          case ukDelete:
            UpdateTable->Delete();
            break;
        }
    }
    UpdateAction = uaApplied;
}
```

See Also

Applying Cached Updates Using a Database (see page 1687)

Applying Cached Updates with Dataset Component Methods (see page 1695)

Handling Cached Update Errors (see page 1696)

3.2.3.7.9.10 Applying Cached Updates with Dataset Component Methods

You can apply updates for individual BDE-enabled datasets directly using the dataset's ApplyUpdates and CommitUpdates methods. Each of these methods encapsulate one phase of the update process:

1. *ApplyUpdates* writes cached changes to a database (phase 1).
2. *CommitUpdates* clears the internal cache when the database write is successful (phase 2).

The following code illustrates how you apply updates within a transaction for the *CustomerQuery* dataset:

```
procedure TForm1.ApplyButtonClick(Sender: TObject)
begin
  Database1.StartTransaction;
  try
    if not (Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
      Database1.TransIsolation := tiDirtyRead;
    CustomerQuery.ApplyUpdates;                { try to write the updates to the database }
    Database1.Commit;                                    { on success, commit the changes }
  except
    Database1.Rollback;                                    { on failure, undo any changes }
    raise;                 { raise the exception again to prevent a call to CommitUpdates }
  end;
  CustomerQuery.CommitUpdates;                       { on success, clear the internal cache }
end;
void __fastcall TForm1::ApplyButtonClick(TObject *Sender)
{
  Database1->StartTransaction();
  try
  {
    if (!Database1->IsSQLBased && Database1->TransIsolation != tiDirtyRead)
      Database1->TransIsolation = tiDirtyRead;
    CustomerQuery->ApplyUpdates(); // try to write the updates to the database
    Database1->Commit(); // on success, commit the changes
  }
  catch (...)
  {
    Database1->Rollback(); // on failure, undo any changes
    throw; // throw the exception again to prevent a call to CommitUpdates
  }
  CustomerQuery->CommitUpdates(); // on success, clear the internal cache
}
```

If an exception is raised during the *ApplyUpdates* call, the database transaction is rolled back. Rolling back the transaction

ensures that the underlying database table is not changed. The raise statement inside the **try...except** block reraises the exception, thereby preventing the call to *CommitUpdates*. Because *CommitUpdates* is not called, the internal cache of updates is not cleared so that you can handle error conditions and possibly retry the update.

See Also

Handling Cached Update Errors (see page 1696)

3.2.3.7.9.11 Creating SQL Statements for Update Components

To update a record in an associated dataset, an update object uses one of three SQL statements. Each update object can only update a single table, so the object's update statements must each reference the same base table.

The three SQL statements delete, insert, and modify records cached for update. You must provide these statements as update object's *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties. You can provide these values at design time or at runtime. For example, the following code specifies a value for the *DeleteSQL* property at runtime:

```
with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
UpdateSQL->DeleteSQL->Clear();
UpdateSQL->DeleteSQL->Add("DELETE FROM Inventory I");
UpdateSQL->DeleteSQL->Add("WHERE (I.ItemNo = :OLD_ItemNo)");
```

At design time, you can use the Update SQL editor (see page 1691) to help you compose the SQL statements that apply updates.

Update objects provide automatic parameter binding (see page 1689) for parameters that reference the dataset's original and updated field values. Typically, therefore, you insert parameters with specially formatted names when you compose the SQL statements (see page 1687).

See Also

Using Multiple Update Objects (see page 1641)

Executing the SQL Statements (see page 1643)

3.2.3.7.9.12 Handling Cached Update Errors

The Borland Database Engine (BDE) specifically checks for user update conflicts and other conditions when attempting to apply updates, and reports any errors. The dataset component's OnUpdateError event enables you to catch and respond to errors. You should create a handler for this event if you use cached updates. If you do not, and an error occurs, the entire update operation fails.

Here is the skeleton code for an *OnUpdateError* event handler:

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { ... perform update error handling here ... }
end;
void __fastcall TForm1::DataSetUpdateError(TDataSet *DataSet,
  EDatabaseError *E, TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  // Respond to errors here...
}
```

DataSet references the dataset to which updates are applied. You can use this dataset to access new and old values during error handling. The original values for fields in each record are stored in a read-only *TField* property called *OldValue*. Changed values are stored in the analogous *TField* property *NewValue*. These values provide the only way to inspect and change update

values in the event handler.

Warning: Do not call any dataset methods that change the current record (such as *Next* and *Prior*). Doing so causes the event handler to enter an endless loop.

The *E* parameter is usually of type *EDBEngineError*. From this exception type, you can extract an error message that you can display to users in your error handler. For example, the following code could be used to display the error message in the caption of a dialog box:

```
ErrorLabel.Caption := E.Message;
ErrorLabel->Caption = E->Message;
```

This parameter is also useful for determining the actual cause of the update error. You can extract specific error codes from *EDBEngineError*, and take appropriate action based on it.

The *UpdateKind* parameter describes the type of update that generated the error. Unless your error handler takes special actions based on the type of update being carried out, your code probably will not make use of this parameter.

The following table lists possible values for *UpdateKind*:

UpdateKind values

Value	Meaning
ukModify	Editing an existing record caused an error.
ukInsert	Inserting a new record caused an error.
ukDelete	Deleting an existing record caused an error.

UpdateAction tells the BDE how to proceed with the update process when your event handler exits. When your update error handler is first called, the value for this parameter is always set to *uaFail*. Based on the error condition for the record that caused the error and what you do to correct it, you typically set *UpdateAction* to a different value before exiting the handler:

- If your error handler can correct the error condition that caused the handler to be invoked, set *UpdateAction* to the appropriate action to take on exit. For error conditions you correct, set *UpdateAction* to *uaRetry* to apply the update for the record again.
- When set to *uaSkip*, the update for the row that caused the error is skipped, and the update for the record remains in the cache after all other updates are completed.
- Both *uaFail* and *uaAbort* cause the entire update operation to end. *uaFail* raises an exception and displays an error message. *uaAbort* raises a silent exception (does not display an error message).

The following code shows an *OnUpdateError* event handler that checks to see if the update error is related to a key violation, and if it is, it sets the *UpdateAction* parameter to *uaSkip*:

```
{ Add 'Bde' to your uses clause for this example }
if (E is EDBEngineError) then
  with EDBEngineError(E) do begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip                    { key violation, just skip this record }
    else
      UpdateAction := uaAbort;              { don't know what's wrong, abort the update }
  end;
// include BDE.hpp in your unit file for this example
void __fastcall TForm1::DataSetUpdateError(TDataSet *DataSet,
  EDatabaseError *E, TUpdateKind UpdateKind, TUpdateAction &UpdateAction)
{
  UpdateAction = uaFail // initialize to fail the update
  if (E->ClassNameIs("EDBEngineError"))
  {
    EDBEngineError *pDBE = (EDBEngineError *)E;
    if (pDBE->Errors[pDBE->ErrorCount - 1]->ErrorCode == DBIERR_KEYVIOL)
      UpdateAction = uaSkip; // Key violation, just skip this record
  }
}
```

Note: If an error occurs during the application of cached updates, an exception is raised and an error message displayed. Unless the *ApplyUpdates* is called from within a try...except construct, an error message to the user displayed from inside your *OnUpdateError* event handler may cause your application to display the same error message twice. To prevent error message duplication, set *UpdateAction* to *uaAbort* to turn off the system-generated error message display.

See Also

Applying Cached Updates with a Database Component Method (see page 1695)

Applying Cached Updates with Dataset Component Methods (see page 1695)

3.2.3.7.10 Using transactions with the BDE

The Borland Database Engine (BDE) has been deprecated, so it will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

Topics

Name	Description
Using Local Transactions (see page 1698)	The BDE supports local transactions against Paradox, dBASE, Access, and FoxPro tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server. **Note:** When using transactions with local Paradox, dBASE, Access, and FoxPro tables, set *TransIsolation* to *tiDirtyRead* instead of using the default value of *tiReadCommitted*. A BDE error is returned if *TransIsolation* is set to anything but *tiDirtyRead* for local tables. When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for... more (see page 1698)
Using Passthrough SQL (see page 1699)	With passthrough SQL, you use a *TQuery*, *TStoredProc*, or *TUpdateSQL* component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard. To use passthrough SQL to control a transaction, you must • Install the proper SQL Links drivers. If you chose the "Typical" installation when installing Delphi, all SQL Links drivers are already properly installed. • Configure your network protocol. See your network administrator for more... more (see page 1699)
Using Transactions with the BDE (see page 1699)	By default, the BDE provides implicit transaction control for your applications. When an application is under implicit transaction control, a separate transaction is used for each record in a dataset that is written to the underlying database. Implicit transactions guarantee both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance. Also, implicit transaction control will not protect logical operations that span more than one... more (see page 1699)

3.2.3.7.10.1 Using Local Transactions

The BDE supports local transactions against Paradox, dBASE, Access, and FoxPro tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server.

Note: When using transactions with local Paradox, dBASE, Access, and FoxPro tables, set *TransIsolation* to *tiDirtyRead* instead of using the default value of *tiReadCommitted*. A BDE error is returned if *TransIsolation* is set to anything but *tiDirtyRead* for local tables.

When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for a record. When a transaction is active, records that are updated are locked until the transaction is

committed or rolled back. On rollback, old record buffers are applied against updated records to restore them to their pre-update states.

Local transactions are more limited than transactions against SQL servers or ODBC drivers. In particular, the following limitations apply to local transactions:

- Automatic crash recovery is not provided.
- Data definition statements are not supported.
- Transactions cannot be run against temporary tables.
- *TransIsolation* level must only be set to *tiDirtyRead*.
- For Paradox, local transactions can only be performed on tables with valid indexes. Data cannot be rolled back on Paradox tables that do not have indexes.
- Only a limited number of records can be locked and modified. With Paradox tables, you are limited to 255 records. With dBASE the limit is 100.
- Transactions cannot be run against the BDE ASCII driver.
- Closing a cursor on a table during a transaction rolls back the transaction unless:
- Several tables are open.
- The cursor is closed on a table to which no changes were made.

See Also

Using Passthrough SQL (see page 1699)

3.2.3.7.10.2 Using Passthrough SQL

With passthrough SQL, you use a *TQuery*, *TStoredProc*, or *TUpdateSQL* component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard.

To use passthrough SQL to control a transaction, you must

- Install the proper SQL Links drivers. If you chose the "Typical" installation when installing Delphi, all SQL Links drivers are already properly installed.
- Configure your network protocol. See your network administrator for more information.
- Have access to a database on a remote server.
- Set SQLPASSTHRU MODE to NOT SHARED using the SQL Explorer. SQLPASSTHRU MODE specifies whether the BDE and passthrough SQL statements can share the same database connections. In most cases, SQLPASSTHRU MODE is set to SHARED AUTOCOMMIT. However, you can't share database connections when using transaction control statements.

  Note: When SQLPASSTHRU MODE is NOT SHARED, you must use separate database components for datasets that pass SQL transaction statements to the server and datasets that do not.

See Also

Using Local Transactions (see page 1698)

3.2.3.7.10.3 Using Transactions with the BDE

By default, the BDE provides implicit transaction control for your applications. When an application is under implicit transaction control, a separate transaction is used for each record in a dataset that is written to the underlying database. Implicit transactions guarantee both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance. Also, implicit transaction control will not protect logical operations that span more than one record.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop applications in a multi-user environment, particularly when your applications run against a remote SQL server, you should control transactions explicitly.

There are two mutually exclusive ways to control transactions explicitly in a BDE-based database application:

- Use the database component to control transactions. The main advantage to using the methods and properties of a database component is that it provides a clean, portable application that is not dependent on a particular database or server. This type of transaction control is supported by all database connection components, and described in Managing transactions (see page 1498).
- Use passthrough SQL (see page 1699) in a query component to pass SQL statements directly to remote SQL or ODBC servers. The main advantage to passthrough SQL is that you can use the advanced transaction management capabilities of a particular database server, such as schema caching. To understand the advantages of your server's transaction management model, see your database server documentation.

When working with local databases, you can only use the database component to create explicit transactions (local databases do not support passthrough SQL). However, there are limitations to using local transactions. For more information on using local transactions, see Using Local Transactions (see page 1698).

Note: You can minimize the number of transactions you need by caching updates. For more information about cached updates, see Using a Client Dataset to Cache Updates (see page 1731).

See Also

BDE-based Architecture (see page 1638)

Using TBatchMove (see page 1683)

Tools for Working with the BDE (see page 1681)

Transactions (see page 1570)

The Data Dictionary (see page 1680)

3.2.3.8 Using client datasets

Topics

Name	Description
Adding a New Index (see page 1709)	There are three ways to add indexes to a client dataset:
Adding Application-specific Information to the Data (see page 1709)	Application developers can add custom information to the client dataset's *Data* property. Because this information is bundled with the data packet, it is included when you save the data to a file or stream (see page 1740). It is copied when you copy the data to another dataset (see page 1714). Optionally, it can be included with the *Delta* property so that a provider can read this information when it receives updates from the client dataset. To save application-specific information with the *Data* property, use the SetOptionalParam method. This method lets you store an OleVariant that contains the data under a specific name. To... more (see page 1709)
Aggregating over groups of records (see page 1710)	By default, maintained aggregates are calculated so that they summarize all the records in the client dataset. However, you can specify that you want to summarize over the records in a group instead. This lets you provide intermediate summaries such as subtotals for groups of records that share a common field value. Before you can specify a maintained aggregate over a group of records, you must use an index that supports the appropriate grouping (see page 1734). Once you have an index that groups the data in the way you want it summarized, specify the *IndexName* and *GroupingLevel* properties of the aggregate... more (see page 1710)

Applying Updates (see page 1711)	Changes made to the client dataset's local copy of data are not sent to the database server (or XML document) until the client application calls the ApplyUpdates method. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called Delta) to the provider. (Note that, when using most client datasets, the provider is internal to the client dataset.) *ApplyUpdates* takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the provider should tolerate before aborting the update process. If *MaxErrors* is *0*, then as soon as an update error occurs,... more (see page 1711)
Choosing the Type of Dataset for Caching Updates (see page 1711)	Delphi includes some specialized client dataset components for caching updates. Each client dataset is associated with a particular data access mechanism (see page 1572). These are listed in the following table: ***Specialized client datasets for caching updates***
Communicating with Providers Using Custom Events (see page 1712)	Client datasets communicate with a provider component through a special interface called IAppServer. If the provider is local, *IAppServer* is the interface to an automatically-generated object that handles all communication between the client dataset and its provider. If the provider is remote, *IAppServer* is the interface to a remote data module on the application server, or (in the case of a SOAP server) an interface generated by the connection component. *TClientDataSet* provides many opportunities for customizing the communication that uses the *IAppServer* interface. Before and after every *IAppServer* method call that is directed at the client dataset's provider, *TClientDataSet* receives... more (see page 1712)
Constraining Data Values (see page 1713)	Client datasets can enforce constraints on the edits a user makes to data. These constraints are applied when the user tries to post changes to the change log. You can always supply custom constraints. These let you provide your own, application-defined limits on what values users post to a client dataset. In addition, when client datasets represent server data that is accessed using the BDE, they also enforce data constraints imported from the database server. If the client dataset works with an external provider (see page 1732) component, the provider can control whether those constraints are sent to the client dataset, and the... more (see page 1713)
Copying Data from Another Dataset (see page 1714)	To copy the data from another dataset at design time, right click the client dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one whose data and structure you want to copy and choose OK. When you copy the source dataset, your client dataset is automatically activated. To copy from another dataset at runtime, you can assign its data directly (see page 1738) or, if the source is another client dataset, you can clone the cursor (see page 1736).
Creating a New Dataset (see page 1714)	There are three ways to define and create client datasets that do not represent server data: • You can define and create a new client dataset using persistent fields or field and index definitions. This follows the same scheme as creating any table type dataset. See Creating and deleting tables (see page 1587) for details. • You can copy an existing dataset (see page 1714) (at design or runtime). • You can create a client dataset from an arbitrary XML document. See Converting XML documents into data packets (see page 1841) for details. Once the dataset is created, you can save it to a file. From then on, you do not need... more (see page 1714)
Deleting and Switching Indexes (see page 1715)	To remove an index you created for a client dataset, call DeleteIndex and specify the name of the index to remove. You cannot remove the DEFAULT_ORDER and CHANGEINDEX indexes. To use a different index when more than one index is available, use the IndexName property to select the index to use. At design time, you can select from available indexes in *IndexName* property drop-down box in the **Object Inspector**.

Editing Data (see page 1715)	Client datasets represent their data as an in-memory data packet. This packet is the value of the client dataset's Data property. By default, however, edits are not stored in the *Data* property. Instead the insertions, deletions, and modifications (made by users or programmatically) are stored in an internal change log, represented by the Delta property. Using a change log serves two purposes: • The change log is required for applying updates to a database server or external provider component. • The change log provides sophisticated support for undoing changes. The LogChanges property lets you disable logging. When *LogChanges* is *True*, changes... more (see page 1715)
Getting Parameters from the Application Server (see page 1715)	There are two circumstances when the client dataset needs to fetch parameter values: • The application needs the value of output parameters on a stored procedure. • The application wants to initialize the input parameters of a query or stored procedure to the current values on the source dataset. Client datasets store parameter values in their Params property. These values are refreshed with any output parameters when the client dataset fetches data from the source dataset (see page 1725). However, there may be times a *TClientDataSet* component in a client application needs output parameters when it is not fetching data. To fetch output parameters... more (see page 1715)
Handling Constraints from the Server (see page 1716)	When a database server defines constraints on what data is valid, it is useful if the client dataset knows about them. That way, the client dataset can ensure that user edits never violate those server constraints. As a result, such violations are never passed to the database server where they would be rejected. This means fewer updates generate error conditions during the updating process. Regardless of the source of data, you can duplicate such server constraints by explicitly adding them to the client dataset. This process is described in Constraining data values (see page 1713). It is more convenient, however, if the... more (see page 1716)
Indicating What Records Are Modified (see page 1717)	While the user edits a client dataset, you may find it useful to provide feedback about the edits that have been made. This is especially useful if you want to allow the user to undo specific edits, for example, by navigating to them and clicking an "Undo" button. The *UpdateStatus* method and *StatusFilter* properties are useful when providing feedback on what updates have occurred: UpdateStatus indicates what type of update, if any, has occurred for the current record. It can be any of the following values: • *usUnmodified* indicates that the current record is unchanged. • *usModified* indicates that the current record... more (see page 1717)
Intervening as Updates Are Applied (see page 1718)	When a client dataset applies its updates, the provider determines how to handle writing the insertions, deletions, and modifications to the database server or source dataset. When you use *TClientDataSet* with an external provider component, you can use the properties and events of that provider to influence the way updates are applied. These are described in Responding to client update requests (see page 1808). When the provider is internal, however, as it is for any client dataset associated with a data access mechanism, you can't set its properties or provide event handlers. As a result, the client dataset publishes one property and... more (see page 1718)

Limiting Records with Parameters (see page 1720)	If the client dataset is • a *TClientDataSet* instance whose associated provider represents a *TTable* or *TSQLTable* component • a *TSimpleDataSet or a TBDEClientDataSet* instance whose *CommandType* property is *ctTable* then it can use the Params property to limit the records that it caches in memory. Each parameter represents a field value that must be matched before a record can be included in the client dataset's data. This works much like a filter, except that with a filter, the records are still cached in memory, but unavailable. Each parameter name must match the name of a field. When using *TClientDataSet*, these... more (see page 1720)
Merging Changes into Data (see page 1720)	When you edit the data in a client dataset, all edits to the data exist only in an in-memory change log. This log can be maintained separately from the data itself, although it is completely transparent to objects that use the client dataset. That is, controls that navigate the client dataset or display its data see a view of the data that includes the changes. If you do not want to back out of changes, however, you should merge the change log into the data of the client dataset by calling the MergeChangeLog method. *MergeChangeLog* overwrites records in *Data* with... more (see page 1720)
Navigating Data in Client Datasets (see page 1721)	If an application uses standard data-aware controls, then a user can navigate through a client dataset's records using the built-in behavior of those controls. You can also navigate programmatically through records using standard dataset methods such as *First*, *Last*, *Next*, and *Prior*. For more information about these methods, see Navigating datasets (see page 1634). Unlike most datasets, client datasets can also position the cursor at a specific record in the dataset by using the RecNo property. Ordinarily an application uses *RecNo* to determine the record number of the current record. Client datasets can, however, set *RecNo* to a... more (see page 1721)
Obtaining Aggregate Values (see page 1721)	To get the value of a maintained aggregate, call the Value method of the *TAggregate* object that represents the aggregate. *Value* returns the maintained aggregate for the group that contains the current record of the client dataset. When you are summarizing over the entire client dataset, you can call *Value* at any time to obtain the maintained aggregate. However, when you are summarizing over grouped information, you must be careful to ensure that the current record is in the group whose summary you want. Because of this, it is a good idea to obtain aggregate values at clearly specified times,... more (see page 1721)
Overriding the Dataset On the Application Server (see page 1722)	The client datasets that are associated with a particular data access mechanism use the CommandText and *Com*mandType properties to specify the data they represent. When using *TClientDataSet*, however, the data is specified by the source dataset, not the client dataset. Typically, this source dataset has a property that specifies an SQL statement to generate the data or the name of a database table or stored procedure. If the provider allows, *TClientDataSet* can override the property on the source dataset that indicates what data it represents. That is, if the provider permits, the client dataset's CommandText property replaces the... more (see page 1722)

Overview of Using Cached Updates (see page 1723)	To use cached updates, the following order of processes must occur in an application: **Indicate the data you want to edit.** How you do this depends on the type of client dataset you are using: • If you are using *TClientDataSet*, Specify the provider component (see page 1729) that represent the data you want to edit. • If you are using a client dataset associated with a particular data access mechanism, you must • Identify the database server by setting the *DBConnection* property to an appropriate connection component. • Indicate what data you want to see by specifying the CommandText and *CommandType* properties. *CommandType* indicates whether... more (see page 1723)
Passing Parameters to the Source Dataset (see page 1723)	Client datasets can pass parameters to the source dataset to specify what data they want provided in the data packets it sends. These parameters can specify • Input parameter values for a query or stored procedure that is run on the application server (see page 1727) • Field values that limit the records sent in data packets (see page 1720) You can specify parameter values that your client dataset sends to the source dataset at design time or at runtime. At design time, select the client dataset and double-click the Params property in the **Object Inspector**. This brings up the collection editor, where you can add, delete,... more (see page 1723)
Refreshing Records (see page 1724)	Client datasets work with an in-memory snapshot of the data from the source dataset. If the source dataset represents server data, then as time elapses other users may modify that data. The data in the client dataset becomes a less accurate picture of the underlying data. Like any other dataset, client datasets have a Refresh method that updates its records to match the current values on the server. However, calling *Refresh* only works if there are no edits in the change log. Calling *Refresh* when there are unapplied edits results in an exception. Client datasets can also update the data... more (see page 1724)
Representing Calculated Values (see page 1725)	As with any dataset, you can add calculated fields (see page 1872) to your client dataset. These are fields whose values you calculate dynamically, usually based on the values of other fields in the same record. Client datasets, however, let you optimize when fields are calculated by using internally calculated fields (see page 1735). You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates (see page 1735).
Requesting Data from the Source Dataset or Document (see page 1725)	Client datasets can control how they fetch their data packets from a provider. By default, they retrieve all records from the source dataset. This is true whether the source dataset and provider are internal components (as with *TBDEClientDataSet,TSimpleDataSet,* and *TIBClientDataSet*), or separate components that supply the data for *TClientDataSet*. You can change how the client dataset fetches records using the *PacketRecords* and *FetchOnDemand* properties.
Saving Changes (see page 1726)	Client datasets use different mechanisms for incorporating changes from the change log, depending on whether the client datasets stores its data in a file or represents data obtained through a provider. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated. File-based applications can simply merge the changes into the local cache represented by the *Data* property. They do not need to worry about resolving local edits with changes made by other users. To merge the change log into the *Data* property, call the *MergeChangeLog* method. Merging changes into data (see page 1720) describes this process. You... more (see page 1726)

Sending Query or Stored Procedure Parameters (see page 1727)	When the client dataset's *CommandType* property is *ctQuery* or *ctStoredProc*, or, if the client dataset is a *TClientDataSet* instance, when the associated provider represents the results of a query or stored procedure, you can use the Params property to specify parameter values. When the client dataset requests data from the source dataset (see page 1725) or uses its Execute method to run a query or stored procedure that does not return a dataset, it passes these parameter values along with the request for data or the execute command. When the provider receives these parameter values, it assigns them to its associated dataset.... more (see page 1727)
Setting Up a Simple Dataset (see page 1727)	Setting up a simple dataset requires two essential steps. Set up: 1. The connection information. 2. The dataset information. The following steps describe setting up a simple dataset in more detail.
Sorting and Indexing (see page 1728)	Using indexes provides several benefits to your applications: • They allow client datasets to locate data quickly. • They let you apply ranges to limit the available records. • They let your application set up relationships with other datasets such as lookup tables or master/detail forms. • They specify the order in which records appear. If a client dataset represents server data or uses an external provider, it inherits a default index and sort order based on the data it receives. The default index is called DEFAULT_ORDER. You can use this ordering, but you cannot change or delete the index. In addition to the... more (see page 1728)
Specifying a Provider (see page 1729)	Unlike the client datasets that are associated with a data access mechanism, *TClientDataSet* has no internal provider component to package data or apply updates. If you want it to represent data from a source dataset or XML document, therefore, you must associated the client dataset with an external provider component. The way you associate *TClientDataSet* with a provider depends on whether the provider is in the same application as the client dataset or on a remote application server running on another system.
Specifying Aggregates (see page 1730)	To specify that you want to calculate summaries over the records in a client dataset, use the Aggregates property. *Aggregates* is a collection of aggregate specifications (TAggregate). You can add aggregate specifications to your client dataset using the Collection Editor at design time, or using the Add method of *Aggregates* at runtime. If you want to create field components for the aggregates, create persistent fields for the aggregated values in the Fields Editor. **Note:** When you create aggregated fields, the appropriate aggregate objects are added to the client dataset's *Aggregates* property automatically. Do not add them explicitly when creating aggregated (see page 1856)... more (see page 1730)
Undoing Changes (see page 1730)	Even though a record's original version remains unchanged in *Data*, each time a user edits a record, leaves it, and returns to it, the user sees the last changed version of the record. If a user or application edits a record a number of times, each changed version of the record is stored in the change log as a separate entry. Storing each change to a record makes it possible to support multiple levels of undo operations should it be necessary to restore a record's previous state: • To remove the last change to a record, call UndoLastChange. *UndoLastChange* takes... more (see page 1730)

Updating Records (see page 1731)	The contents of the change log are stored as a data packet in the client dataset's *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database (or source dataset or XML document). When a client applies updates to the server, the following steps occur: 1. The client application calls the ApplyUpdates method (see page 1711) of a client dataset object. This method passes the contents of the client dataset's *Delta* property to the (internal or external) provider. *Delta* is a data packet that contains a client dataset's updated, inserted, and deleted records. 2. The provider applies the... more (see page 1731)
Using a Client Dataset to Cache Updates (see page 1731)	By default, when you edit data in most datasets, every time you delete or post a record, the dataset generates a transaction, deletes or writes that record to the database server, and commits the transaction. If there is a problem writing changes to the database, your application is notified immediately: the dataset raises an exception when you post the record. If your dataset uses a remote database server, this approach can degrade performance due to network traffic between your application and the server every time you move to a new record after editing the current record. To minimize the network... more (see page 1731)
Using a Client Dataset with a Provider (see page 1732)	A client dataset uses a provider to supply it with data and apply updates when • It caches updates from a database server or another dataset. • It represents the data in an XML document. • It stores the data in the client portion of a multi-tiered application. For any client dataset other than *TClientDataSet*, this provider is internal, and so not directly accessible by the application. With *TClientDataSet*, the provider is an external component that links the client dataset to an external source of data. An external provider component can reside in the same application as the client dataset, or... more (see page 1732)
Using a Client Dataset with File-based Data (see page 1733)	Client datasets can work with dedicated files on disk as well as server data. This allows them to be used in file-based database applications (see page 1564) and "briefcase model" applications (see page 1565). The special files that client datasets use for their data are called MyBase. **Tip:** All client datasets are appropriate for a briefcase model application, but for a pure MyBase application (one that does not use a provider), it is preferable to use *TClientDataSet*, because it involves less overhead. In a pure MyBase application, the client application cannot get table definitions and data from the server, and there is no server... more (see page 1733)
Using a Simple Dataset (see page 1733)	*TSimpleDataSet* is a special type of client dataset designed for simple two-tiered applications. Like a unidirectional dataset, it can use an SQL connection component to connect to a database server and specify an SQL statement to execute on that server. Like other client datasets, it buffers data in memory to allow full navigation and editing support. *TSimpleDataSet* works the same way as a generic client dataset (*TClientDataSet*) that is linked to a unidirectional dataset by a dataset provider. In fact, *TSimpleDataSet* has its own, internal provider, which it uses to communicate with an internally created unidirectional dataset. Using... more (see page 1733)
Using Indexes to Group Data (see page 1734)	When you use an index in your client dataset, it automatically imposes a sort order on the records. Because of this order, adjacent records usually contain duplicate values on the fields that make up the index. For example, consider the following fragment from an orders table that is indexed on the SalesRep and Customer fields:

Using Internally Calculated Fields in Client Datasets (see page 1735)	In other datasets, your application must compute the value of calculated fields every time the record changes or the user edits any fields in the current record. It does this in an OnCalcFields event handler. While you can still do this, client datasets let you minimize the number of times calculated fields must be recomputed by saving calculated values in the client dataset's data. When calculated values are saved with the client dataset, they must still be recomputed when the user edits the current record, but your application need not recompute values every time the current record changes. To save... more (see page 1735)
Using Maintained Aggregates (see page 1735)	Client datasets provide support for summarizing data over groups of records. Because these summaries are automatically updated as you edit the data in the dataset, this summarized data is called a "maintained aggregate." In their simplest form, maintained aggregates let you obtain information such as the sum of all values in a column of the client dataset. They are flexible enough, however, to support a variety of summary calculations and to provide subtotals over groups of records defined by a field in an index that supports grouping (see page 1734). The following topics describe how to • Specify aggregates (see page 1730). • Aggregate Over Groups (see page 1710)... more (see page 1735)
When to Use TSimpleDataSet (see page 1736)	*TSimpleDataSet* is intended for use in a simple two-tiered database applications and briefcase model applications. It provides an easy-to-set up component for linking to the database server, fetching data, caching updates, and applying them back to the server. It can be used in most two-tiered applications. There are times, however, when it is more appropriate to use *TClientDataSet*: • If you are not using data from a database server (for example, if you are using a dedicated file on disk (see page 1564)), then *TClientDataSet* has the advantage of less overhead. • Only *TClientDataSet* can be used in a multi-tiered database application .... more (see page 1736)
Cloning a Client Dataset Cursor (see page 1736)	Client datasets use the *CloneCursor* method to let you work with a second view of the data at runtime. *CloneCursor* lets a second client dataset share the original client dataset's data. This is less expensive than copying all the original data, but, because the data is shared, the second client dataset can't modify the data without affecting the original client dataset. *CloneCursor* takes three parameters: *Source* specifies the client dataset to clone. The last two parameters (*Reset* and *KeepSettings*) indicate whether to copy information other than the data. This information includes any filters, the current index, links to... more (see page 1736)
Working with Data Using a Client Dataset (see page 1737)	Like any dataset, you can use client datasets to supply the data for data-aware controls using a data source component. See Using data controls (see page 1778) for information on how to display database information in data-aware controls. Client datasets implement all the properties an methods inherited from *TDataSet*. For a complete introduction to this generic dataset behavior, see Understanding datasets (see page 1632). In addition, client datasets implement many of the features common to table type datasets (see page 1620) such as • Sorting records with indexes (see page 1609). • Using Indexes to search for records (see page 1615). • Limiting records with ranges (see page 1597). • Creating master/detail relationships (see page 1590). • Controlling read/write access (see page 1587)... more (see page 1737)

Reconciling Update Errors (see page 1737)	There are two events that let you handle errors that occur during the update process: • During the update process, the internal provider generates an OnUpdateError event every time it encounters an update that it can't handle. If you correct the problem in an *OnUpdateError* event handler, then the error does not count toward the maximum number of errors passed to the *ApplyUpdates* method. This event only occurs for client datasets that use an internal provider. If you are using *TClientDataSet*, you can use the provider component's *OnUpdateError* event instead. • After the entire update operation is finished, the client dataset... more (see page 1737)
Assigning Data Directly (see page 1738)	You can use the client dataset's *Data* property to assign data to a client dataset from another dataset. *Data* is a data packet in the form of an OleVariant. A data packet can come from another client dataset or from any other dataset by using a provider. Once a data packet is assigned to *Data*, its contents are displayed automatically in data-aware controls connected to the client dataset by a data source component. When you open a client dataset that represents server data or that uses an external provider component, data packets are automatically assigned to *Data*. When... more (see page 1738)
Loading Data from a File or Stream (see page 1739)	To load data from a file, call a client dataset's LoadFromFile method. *LoadFromFile* takes one parameter, a string that specifies the file from which to read data. The file name can be a fully qualified path name, if appropriate. If you always load the client dataset's data from the same file, you can use the FileName property instead. If *FileName* names an existing file, the data is automatically loaded when the client dataset is opened. To load data from a stream, call the client dataset's *LoadFromStream* method. *LoadFromStream* takes one parameter, a stream object that supplies the data. The data... more (see page 1739)
Saving Data to a File or Stream (see page 1740)	Even when you have merged changes into the data of a client dataset, this data still exists only in memory. While it persists if you close the client dataset and reopen it in your application, it will disappear when your application shuts down. To make the data permanent, it must be written to disk. Write changes to disk using the SaveToFile method. *SaveToFile* takes one parameter, a string that specifies the file into which to write data. The file name can be a fully qualified path name, if appropriate. If the file already exists, its current contents are completely overwritten.... more (see page 1740)
Using Client Datasets: Overview (see page 1740)	Client datasets are specialized datasets that hold all their data in memory. The support for manipulating the data they store in memory is provided by midaslib.dcu or midas.dll. The format client datasets use for storing data is self-contained and easily transported, which allows client datasets to • Read from and write to dedicated files on disk, acting as a file-based dataset. Properties and methods supporting this mechanism are described in Using a client dataset with file-based data (see page 1733). • Cache updates for data from a database server. Client dataset features that support cached updates are described in Using a client dataset to (see page 1731)... more (see page 1740)
Limiting What Records Appear (see page 1741)	To restrict users to a subset of available data on a temporary basis, applications can use ranges and filters. When you apply a range or a filter, the client dataset does not display all the data in its in-memory cache. Instead, it only displays the data that meets the range or filter conditions. For more information about using filters, see Displaying and editing a subset of data using filters (see page 1631). For more information about ranges, see Limiting records with ranges (see page 1597). With most datasets, filter strings are parsed into SQL commands that are then implemented on the database server. Because... more (see page 1741)

3.2.3.8.1 Adding a New Index

There are three ways to add indexes to a client dataset:

Methods	Description
Use the IndexFieldNames property	To create a temporary index at runtime that sorts the records in the client dataset, you can use the IndexFieldNames property. Specify field names, separated by semicolons. Ordering of field names in the list determines their order in the index. This is the least powerful method of adding indexes. You can't specify a descending or case-insensitive index, and the resulting indexes do not support grouping (see page 1734). These indexes do not persist when you close the dataset, and are not saved when you save the client dataset to a file.
Call AddIndex	To create an index at runtime that can be used for grouping, call AddIndex. *AddIndex* lets you specify the properties of the index, including: The name of the index. This can be used for switching indexes (see page 1715) at runtime. The fields that make up the index. The index uses these fields to sort records and to locate records that have specific values on these fields. How the index sorts records. By default, indexes impose an ascending sort order (based on the machine's locale). This default sort order is case-sensitive. You can set options to make the entire index case-insensitive or to sort in descending order. Alternately, you can provide a list of fields to be sorted case-insensitively and a list of fields to be sorted in descending order. The default level of grouping support (see page 1734) for the index. Indexes created with *AddIndex* do not persist when the client dataset is closed. (That is, they are lost when you reopen the client dataset). You can"t call *AddIndex* when the dataset is closed. Indexes you add using *AddIndex* are not saved when you save the client dataset to a file.
Use the IndexDefs property	The third way to create an index is at the time the client dataset is created. Before creating the client dataset, specify the desired indexes using the *IndexDefs* property. The indexes are then created along with the underlying dataset when you call CreateDataSet. See Creating and deleting tables (see page 1587) for more information about creating client datasets. As with *AddIndex*, indexes you create with the dataset support grouping, can sort in ascending order on some fields and descending order on others, and can be case insensitive on some fields and case sensitive on others.

Tip: You can index and sort on internally calculated fields with client datasets.

See Also

Using Maintained Aggregates (see page 1735)

Deleting and Switching Indexes (see page 1715)

Using Indexes to Group Data (see page 1734)

3.2.3.8.2 Adding Application-specific Information to the Data

Application developers can add custom information to the client dataset's *Data* property. Because this information is bundled with the data packet, it is included when you save the data to a file or stream (see page 1740). It is copied when you copy the data to another dataset (see page 1714). Optionally, it can be included with the *Delta* property so that a provider can read this information when it receives updates from the client dataset.

To save application-specific information with the *Data* property, use the SetOptionalParam method. This method lets you store an OleVariant that contains the data under a specific name.

To retrieve this application-specific information, use the GetOptionalParam method, passing in the name that was used when the information was stored.

See Also

Navigating Data in Client Datasets (see page 1721)

Limiting What Records Appear (see page 1741)

Constraining Data Values (see page 1713)

Editing Data (see page 1715)

Sorting and Indexing (see page 1728)

Representing Calculated Values (see page 1725)

Adding Custom Information to Data Packets (see page 1808)

Copying Data from Another Dataset (see page 1714)

3.2.3.8.3 Aggregating over groups of records

By default, maintained aggregates are calculated so that they summarize all the records in the client dataset. However, you can specify that you want to summarize over the records in a group instead. This lets you provide intermediate summaries such as subtotals for groups of records that share a common field value. Before you can specify a maintained aggregate over a group of records, you must use an index that supports the appropriate grouping (see page 1734).

Once you have an index that groups the data in the way you want it summarized, specify the *IndexName* and *GroupingLevel* properties of the aggregate to indicate what index it uses, and which group or subgroup on that index defines the records it summarizes.

For example, consider the following fragment from an orders table that is grouped by SalesRep and, within SalesRep, by Customer:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

The following code sets up a maintained aggregate that indicates the total amount for each sales representative:

```
Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';
Agg->Expression = "Sum(Amount)";
Agg->IndexName = "SalesCust";
Agg->GroupingLevel = 1;
Agg->AggregateName = "Total for Rep";
```

To add an aggregate that summarizes for each customer within a given sales representative, create a maintained aggregate with level 2.

Maintained aggregates that summarize over a group of records are associated with a specific index. The *Aggregates* property can include aggregates that use different indexes. However, only the aggregates that summarize over the entire dataset and those that use the current index are valid. Changing the current index changes which aggregates are valid. To determine which

aggregates are valid at any time, use the *ActiveAggs* property.

See Also

Specifying Aggregates (see page 1730)

Obtaining Aggregate Values (see page 1721)

3.2.3.8.4 Applying Updates

Changes made to the client dataset's local copy of data are not sent to the database server (or XML document) until the client application calls the ApplyUpdates method. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called Delta) to the provider. (Note that, when using most client datasets, the provider is internal to the client dataset.)

ApplyUpdates takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the provider should tolerate before aborting the update process. If *MaxErrors* is *0*, then as soon as an update error occurs, the entire update process is terminated. No changes are written to the database, and the client dataset's change log remains intact. If *MaxErrors* is *-1*, any number of errors is tolerated, and the change log contains all records that could not be successfully applied. If *MaxErrors* is a positive value, and more errors occur than are permitted by *MaxErrors*, all updates are aborted. If fewer errors occur than specified by *MaxErrors*, all records successfully applied are automatically cleared from the client dataset's change log.

ApplyUpdates returns the number of actual errors encountered, which is always less than or equal to *MaxErrors* plus one. This return value indicates the number of records that could not be written to the database.

The client dataset's *ApplyUpdates* method does the following:

- It indirectly calls the provider's ApplyUpdates method. The provider's *ApplyUpdates* method writes the updates to the database, source dataset, or XML document and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset.
- The client dataset 's *ApplyUpdates* method then attempts to reconcile these problem records by calling the Reconcile method. *Reconcile* is an error-handling routine that calls the OnReconcileError event handler. You must code the *OnReconcileError* event handler to correct errors. For details about using *OnReconcileError*, see Reconciling Update Errors (see page 1737).
- Finally, *Reconcile* removes successfully applied changes from the change log and updates Data to reflect the newly updated records. When *Reconcile* completes, *ApplyUpdates* reports the number of errors that occurred.

  Warning: In some cases, the provider can't determine how to apply updates (for example, when applying updates from a stored procedure or multi-table join). Client datasets and provider components generate events that let you handle these situations. See Intervening as updates are applied (see page 1718) for details.

  Tip: If the provider is on a stateless application server, you may want to communicate with it about persistent state information before or after you apply updates. *TClientDataSet* receives a BeforeApplyUpdates event before the updates are sent, which lets you send persistent state information to the server. After the updates are applied (but before the reconcile process), *TClientDataSet* receives an AfterApplyUpdates event where you can respond to any persistent state information returned by the application server.

See Also

Reconciling Update Errors (see page 1737)

Responding to Client Update Requests (see page 1808)

Intervening as Updates Are Applied (see page 1718)

3.2.3.8.5 Choosing the Type of Dataset for Caching Updates

Delphi includes some specialized client dataset components for caching updates. Each client dataset is associated with a particular data access mechanism (see page 1572). These are listed in the following table:

Specialized client datasets for caching updates

Client dataset	Data access mechanism
TSimpleDataSet	dbExpress
TIBClientDataSet	InterBase Express

Generic client dataset for caching updates

Client dataset	Data access mechanism
TClientDataSet	dbExpress, Blackfish SQL, dbGo, BDE

You can cache updates using the generic client dataset (TClientDataSet) with an external provider and source dataset. For information about using *TClientDataSet* with an external provider, see Using a client dataset with a provider (see page 1732).

Note: The specialized client datasets associated with each data access mechanism actually use a provider and source dataset as well. However, both the provider and the source dataset are internal to the client dataset.

It is simplest to use one of the specialized client datasets to cache updates. However, there are times when it is preferable to use *TClientDataSet* with an external provider:

- If you are using a data access mechanism that does not have a specialized client dataset, you must use *TClientDataSet* with an external provider component. For example, if the data comes from an XML document or custom dataset.
- If you are working with tables that are related in a master/detail relationship, you should use *TClientDataSet* and connect it, using a provider, to the master table of two source datasets linked in a master/detail relationship. The client dataset sees the detail dataset as a nested dataset field. This approach is necessary so that updates to master and detail tables can be applied in the correct order.
- If you want to code event handlers that respond to the communication between the client dataset and the provider (for example, before and after the client dataset fetches records from the provider), you must use *TClientDataSet* with an external provider component. The specialized client datasets publish the most important events for applying updates (*OnReconcileError*, *BeforeUpdateRecord* and OnGetTableName), but do not publish the events surrounding communication between the client dataset and its provider, because they are intended primarily for multi-tiered applications.

See Also

Overview of Using Cached Updates (see page 1723)

Updating Records (see page 1731)

Indicating What Records Are Modified (see page 1717)

3.2.3.8.6 Communicating with Providers Using Custom Events

Client datasets communicate with a provider component through a special interface called IAppServer. If the provider is local, *IAppServer* is the interface to an automatically-generated object that handles all communication between the client dataset and its provider. If the provider is remote, *IAppServer* is the interface to a remote data module on the application server, or (in the case of a SOAP server) an interface generated by the connection component.

TClientDataSet provides many opportunities for customizing the communication that uses the *IAppServer* interface. Before and after every *IAppServer* method call that is directed at the client dataset's provider, *TClientDataSet* receives special events that allow it to communicate arbitrary information with its provider. These events are matched with similar events on the provider. Thus for example, when the client dataset calls its *ApplyUpdates* method, the following events occur:

1. The client dataset receives a *BeforeApplyUpdates* event, where it specifies arbitrary custom information in an OleVariant called *OwnerData*.
2. The provider receives a *BeforeApplyUpdates* event, where it can respond to the *OwnerData* from the client dataset and update the value of *OwnerData* to new information.

3. The provider goes through its normal process of assembling a data packet (including all the accompanying events).
4. The provider receives an *AfterApplyUpdates* event, where it can respond to the current value of *OwnerData* and update it to a value for the client dataset.
5. The client dataset receives an *AfterApplyUpdates* event, where it can respond to the returned value of *OwnerData*.

Every other *IAppServer* method call is accompanied by a similar set of *BeforeXXX* and *AfterXXX* events that let you customize the communication between client dataset and provider.

In addition, the client dataset has a special method, DataRequest, whose only purpose is to allow application-specific communication with the provider. When the client dataset calls *DataRequest*, it passes an OleVariant as a parameter that can contain any information you want. This, in turn, generates an is the OnDataRequest event on the provider, where you can respond in any application-defined way and return a value to the client dataset.

See Also

Specifying a Provider (see page 1729)

Requesting Data from the Source Dataset or Document (see page 1725)

Passing Parameters to the Application Server (see page 1723)

Getting Parameters From the Application Server (see page 1715)

Handling Constraints from the Server (see page 1716)

Refreshing Records (see page 1724)

Overriding the Dataset On the Application Server (see page 1722)

3.2.3.8.7 Constraining Data Values

Client datasets can enforce constraints on the edits a user makes to data. These constraints are applied when the user tries to post changes to the change log. You can always supply custom constraints. These let you provide your own, application-defined limits on what values users post to a client dataset.

In addition, when client datasets represent server data that is accessed using the BDE, they also enforce data constraints imported from the database server. If the client dataset works with an external provider (see page 1732) component, the provider can control whether those constraints are sent to the client dataset, and the client dataset can control whether it uses them. For details on how the provider controls whether constraints are included in data packets, see Handling server constraints (see page 1814). For details on how and why client dataset can turn off enforcement of server constraints, see Handling constraints from the server (see page 1716).

Specifying custom constraints

You can use the properties of the client dataset's field components to impose your own constraints on what data users can enter. Each field component has two properties that you can use to specify constraints:

- The DefaultExpression property defines a default value that is assigned to the field if the user does not enter a value. Note that if the database server or source dataset also assigns a default expression for the field, the client dataset's version takes precedence because it is assigned before the update is applied back to the database server or source dataset.
- The CustomConstraint property lets you assign a constraint condition that must be met before a field value can be posted. Custom constraints defined this way are applied in addition to any constraints imported from the server. For more information about working with custom constraints on field components, see Creating a custom constraint (see page 1855).

In addition, you can create record-level constraints using the client dataset's Constraints property. *Constraints* is a collection of TCheckConstraint objects, where each object represents a separate condition. Use the CustomConstraint property of a *TCheckConstraint* object to add your own constraints that are checked when you post records.

See Also

Navigating Data in Client Datasets (see page 1721)

Limiting What Records Appear (see page 1741)

Editing Data (see page 1715)

Sorting and Indexing (see page 1728)

Representing Calculated Values (see page 1725)

Adding Application-specific Information to the Data (see page 1709)

Copying Data from Another Dataset (see page 1714)

3.2.3.8.8 Copying Data from Another Dataset

To copy the data from another dataset at design time, right click the client dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one whose data and structure you want to copy and choose OK. When you copy the source dataset, your client dataset is automatically activated.

To copy from another dataset at runtime, you can assign its data directly (see page 1738) or, if the source is another client dataset, you can clone the cursor (see page 1736).

See Also

Navigating Data in Client Datasets (see page 1721)

Limiting What Records Appear (see page 1741)

Constraining Data Values (see page 1713)

Editing Data (see page 1715)

Sorting and Indexing (see page 1728)

Representing Calculated Values (see page 1725)

Adding Application-specific Information to the Data (see page 1709)

3.2.3.8.9 Creating a New Dataset

There are three ways to define and create client datasets that do not represent server data:

- You can define and create a new client dataset using persistent fields or field and index definitions. This follows the same scheme as creating any table type dataset. See Creating and deleting tables (see page 1587) for details.
- You can copy an existing dataset (see page 1714) (at design or runtime).
- You can create a client dataset from an arbitrary XML document. See Converting XML documents into data packets (see page 1841) for details.

Once the dataset is created, you can save it to a file. From then on, you do not need to recreate the table, only load it from the file you saved. When beginning a file-based database application, you may want to first create and save empty files for your datasets before writing the application itself. This way, you start with the metadata for your client dataset already defined, making it easier to set up the user interface.

See Also

Loading Data from a File or Stream (see page 1739)

Merging Changes into Data (see page 1720)

Saving Data to a File or Stream (see page 1740)

3.2.3.8.10 Deleting and Switching Indexes

To remove an index you created for a client dataset, call DeleteIndex and specify the name of the index to remove. You cannot remove the DEFAULT_ORDER and CHANGEINDEX indexes.

To use a different index when more than one index is available, use the IndexName property to select the index to use. At design time, you can select from available indexes in *IndexName* property drop-down box in the **Object Inspector**.

See Also

Adding a New Index (see page 1709)

Using Indexes to Group Data (see page 1734)

3.2.3.8.11 Editing Data

Client datasets represent their data as an in-memory data packet. This packet is the value of the client dataset's Data property. By default, however, edits are not stored in the *Data* property. Instead the insertions, deletions, and modifications (made by users or programmatically) are stored in an internal change log, represented by the Delta property. Using a change log serves two purposes:

- The change log is required for applying updates to a database server or external provider component.
- The change log provides sophisticated support for undoing changes.

The LogChanges property lets you disable logging. When *LogChanges* is *True*, changes are recorded in the log. When *LogChanges* is *False*, changes are made directly to the *Data* property. You can disable the change log in file-based applications if you do not want the undo support.

Edits in the change log remain there until they are removed by the application. Applications remove edits when

- Undoing changes (see page 1730)
- Saving changes (see page 1726)

  Note: Saving the client dataset to a file does not remove edits from the change log. When you reload the dataset, the *Data* and *Delta* properties are the same as they were when the data was saved.

See Also

Navigating Data in Client Datasets (see page 1721)

Limiting What Records Appear (see page 1741)

Constraining Data Values (see page 1713)

Sorting and Indexing (see page 1728)

Representing Calculated Values (see page 1725)

Adding Application-specific Information to the Data (see page 1709)

Copying Data from Another Dataset (see page 1714)

3.2.3.8.12 Getting Parameters from the Application Server

There are two circumstances when the client dataset needs to fetch parameter values:

- The application needs the value of output parameters on a stored procedure.
- The application wants to initialize the input parameters of a query or stored procedure to the current values on the source dataset.

Client datasets store parameter values in their Params property. These values are refreshed with any output parameters when the client dataset fetches data from the source dataset (see page 1725). However, there may be times a *TClientDataSet* component in a client application needs output parameters when it is not fetching data.

To fetch output parameters when not fetching records, or to initialize input parameters, the client dataset can request parameter values from the source dataset by calling the FetchParams method. The parameters are returned in a data packet from the provider and assigned to the client dataset's *Params* property.

At design time, the *Params* property can be initialized by right-clicking the client dataset and choosing Fetch Params.

Note: There is never a need to call *FetchParams* when the client dataset uses an internal provider and source dataset, because the *Params* property always reflects the parameters of the internal source dataset. With *TClientDataSet*, the *FetchParams* method (or the Fetch Params command) only works if the client dataset is connected to a provider whose associated dataset can supply parameters. For example, if the source dataset is a table type dataset, there are no parameters to fetch.

The *Params* property can also be used to pass parameter values to the source dataset. For details on how to do this, see Passing parameters to the source dataset (see page 1723).

If the provider is on a separate system as part of a stateless application server, you can't use *FetchParams* to retrieve output parameters. In a stateless application server, other clients can change and rerun the query or stored procedure, changing output parameters before the call to *FetchParams*. To retrieve output parameters from a stateless application server, use the Execute method. If the provider is associated with a query or stored procedure, *Execute* tells the provider to execute the query or stored procedure and return any output parameters. These returned parameters are then used to automatically update the *Params* property.

See Also

Specifying a Provider (see page 1729)

Requesting Data from the Source Dataset or Document (see page 1725)

Passing Parameters to the Application Server (see page 1723)

Handling Constraints from the Server (see page 1716)

Refreshing Records (see page 1724)

Communicating with Providers Using Custom Events (see page 1712)

Overriding the Dataset On the Application Server (see page 1722)

3.2.3.8.13 Handling Constraints from the Server

When a database server defines constraints on what data is valid, it is useful if the client dataset knows about them. That way, the client dataset can ensure that user edits never violate those server constraints. As a result, such violations are never passed to the database server where they would be rejected. This means fewer updates generate error conditions during the updating process.

Regardless of the source of data, you can duplicate such server constraints by explicitly adding them to the client dataset. This process is described in Constraining data values (see page 1713).

It is more convenient, however, if the server constraints are automatically included in data packets. Then you need not explicitly specify default expressions and constraints, and the client dataset changes the values it enforces when the server constraints change. By default, this is exactly what happens: if the source dataset is aware of server constraints, the provider automatically includes them in data packets and the client dataset enforces them when the user posts edits to the change log.

Note: Only datasets that use the BDE can import constraints from the server. This means that server constraints are only included in data packets when using *TBDEClientDataSet* or *TClientDataSet* with a provider that represents a BDE-based dataset. For more information on how to import server constraints and how to prevent a provider from including them in data packets, see Handling server constraints (see page 1814).

Note: For more information on working with the constraints once they have been imported, see Using server constraints (see page 1863).

While importing server constraints and expressions is an extremely valuable feature that helps an application preserve data integrity, there may be times when it needs to disable constraints on a temporary basis. For example, if a server constraint is based on the current maximum value of a field, but the client dataset uses incremental fetching, the current maximum value for a field in the client dataset may differ from the maximum value on the database server, and constraints may be invoked differently. In another case, if a client dataset applies a filter to records when constraints are enabled, the filter may interfere in unintended ways with constraint conditions. In each of these cases, an application may disable constraint-checking.

To disable constraints temporarily, call the DisableConstraints method. Each time *DisableConstraints* is called, a reference count is incremented. While the reference count is greater than zero, constraints are not enforced on the client dataset.

To reenable constraints for the client dataset, call the dataset's EnableConstraints method. Each call to *EnableConstraints* decrements the reference count. When the reference count is zero, constraints are enabled again.

Tip: Always call *DisableConstraints* and *EnableConstraints* in paired blocks to ensure that constraints are enabled when you intend them to be.

See Also

Specifying a Provider (see page 1729)

Requesting Data from the Source Dataset or Document (see page 1725)

Passing Parameters to the Application Server (see page 1723)

Getting Parameters From the Application Server (see page 1715)

Refreshing Records (see page 1724)

Communicating with Providers Using Custom Events (see page 1712)

Overriding the Dataset On the Application Server (see page 1722)

3.2.3.8.14 Indicating What Records Are Modified

While the user edits a client dataset, you may find it useful to provide feedback about the edits that have been made. This is especially useful if you want to allow the user to undo specific edits, for example, by navigating to them and clicking an "Undo" button.

The *UpdateStatus* method and *StatusFilter* properties are useful when providing feedback on what updates have occurred:

UpdateStatus indicates what type of update, if any, has occurred for the current record. It can be any of the following values:

- *usUnmodified* indicates that the current record is unchanged.
- *usModified* indicates that the current record has been edited.
- *usInserted* indicates a record that was inserted by the user.
- *usDeleted* indicates a record that was deleted by the user.

StatusFilter controls what type of updates in the change log are visible. *StatusFilter* works on cached records in much the same way as filters work on regular data. *StatusFilter* is a set, so it can contain any combination of the following values:

- *usUnmodified* indicates an unmodified record.
- *usModified* indicates a modified record.
- *usInserted* indicates an inserted record.
- *usDeleted* indicates a deleted record.

By default, *StatusFilter* is the set [*usModified*, *usInserted*, *usUnmodified*]. You can add *usDeleted* to this set to provide feedback

about deleted records as well.

Note: *UpdateStatus* and *StatusFilter* are also useful in *BeforeUpdateRecord* and *OnReconcileError* event handlers. For information about *BeforeUpdateRecord*, see Intervening as updates are applied (see page 1718) For information about *OnReconcileError*, see Reconciling Update Errors. (see page 1737)

The following example shows how to provide feedback about the update status of records using the *UpdateStatus* method. It assumes that you have changed the *StatusFilter* property to include *usDeleted*, allowing deleted records to remain visible in the dataset. It further assumes that you have added a calculated field to the dataset called "Status."

```
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
  with ClientDataSet1 do begin
    case UpdateStatus of
      usUnmodified: FieldByName('Status').AsString := '';
      usModified: FieldByName('Status').AsString := 'M';
      usInserted: FieldByName('Status').AsString := 'I';
      usDeleted: FieldByName('Status').AsString := 'D';
    end;
  end;
end;
void __fastcall TForm1::ClientDataSet1CalcFields(TDataSet *DataSet)
{
  switch (DataSet->UpdateStatus())
  {
    case usUnmodified:
      ClientDataSet1Status->Value = NULL; break;
    case usModified:
      ClientDataSet1Status->Value = "M"; break;
    case usInserted:
      ClientDataSet1Status->Value = "I"; break;
    case usDeleted:
      ClientDataSet1Status->Value = "D"; break;
  }
}
```

See Also

Overview of Using Cached Updates (see page 1723)

Updating Records (see page 1731)

Choosing the Type of Dataset for Caching Updates (see page 1711)

3.2.3.8.15 Intervening as Updates Are Applied

When a client dataset applies its updates, the provider determines how to handle writing the insertions, deletions, and modifications to the database server or source dataset. When you use *TClientDataSet* with an external provider component, you can use the properties and events of that provider to influence the way updates are applied. These are described in Responding to client update requests (see page 1808).

When the provider is internal, however, as it is for any client dataset associated with a data access mechanism, you can't set its properties or provide event handlers. As a result, the client dataset publishes one property and two events that let you influence how the internal provider applies updates.

- UpdateMode controls what fields are used to locate records in the SQL statements the provider generates for applying updates. *UpdateMode* is identical to the provider's *UpdateMode* property. For information on the provider's *UpdateMode* property, see Influencing how updates are applied (see page 1815).
- OnGetTableName lets you supply the provider with the name of the database table to which it should apply updates. This lets the provider generate the SQL statements for updates when it can't identify the database table from the stored procedure or query specified by *CommandText*. For example, if the query executes a multi-table join that only requires updates to a single table, supplying an *OnGetTableName* event handler allows the internal provider to correctly apply updates. An *OnGetTableName* event handler has three parameters: the internal provider component, the internal dataset that fetched the

data from the server, and a parameter to return the table name to use in the generated SQL.

- BeforeUpdateRecord occurs for every record in the delta packet. This event lets you make any last-minute changes before the record is inserted, deleted, or modified. It also provides a way for you to execute your own SQL statements to apply the update in cases where the provider can't generate correct SQL (for example, for multi-table joins where multiple tables must be updated.) A *BeforeUpdateRecord* event handler has five parameters: the internal provider component, the internal dataset that fetched the data from the server, a delta packet that is positioned on the record that is about to be updated, an indication of whether the update is an insertion, deletion, or modification, and a parameter that returns whether the event handler performed the update. The use of these is illustrated in the following event handler. For simplicity, the example assumes the SQL statements are available as global variables that only need field values:

```
procedure TForm1.SimpleDataSet1BeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind: TUpdateKind;
  var Applied Boolean);
var
 SQL: string;
 Connection: TSQLConnection;
begin
 Connection := (SourceDS as TSimpleDataSet).Connection;
 case UpdateKind of
 ukModify:
   begin
    { 1st dataset: update Fields[1], use Fields[0] in where clause }
     SQL := Format(UpdateStmt1, [DeltaDS.Fields[1].NewValue, DeltaDS.Fields[0].OldValue]);
     Connection.Execute(SQL, nil, nil);
    { 2nd dataset: update Fields[2], use Fields[3] in where clause }
     SQL := Format(UpdateStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].OldValue]);
     Connection.Execute(SQL, nil, nil);
   end;
 ukDelete:
   begin
    { 1st dataset: use Fields[0] in where clause }
     SQL := Format(DeleteStmt1, [DeltaDS.Fields[0].OldValue]);
     Connection.Execute(SQL, nil, nil);
    { 2nd dataset: use Fields[3] in where clause }
     SQL := Format(DeleteStmt2, [DeltaDS.Fields[3].OldValue]);
     Connection.Execute(SQL, nil, nil);
   end;
 ukInsert:
   begin
    { 1st dataset: values in Fields[0] and Fields[1] }
      SQL := Format(InsertStmt1, [DeltaDS.Fields[0].NewValue, DeltaDS.Fields[1].NewValue]);
     Connection.Execute(SQL, nil, nil);
    { 2nd dataset: values in Fields[2] and Fields[3] }
     SQL := Format(InsertStmt2, [DeltaDS.Fields[2].NewValue, DeltaDS.Fields[3].NewValue]);
     Connection.Execute(SQL, nil, nil);
   end;
 end;
 Applied := True;
end;
void __fastcall TForm1::SimpleDataSet1BeforeUpdateRecord(TObject *Sender,
   TDataSet *SourceDS, TCustomClientDataSet *DeltaDS, TUpdateKind UpdateKind, bool &Applied)
{
  TSQLConnection *pConn := (dynamic_cast<TSimpleDataSet *>(SourceDS)->Connection);
  char buffer[256];
  switch (UpdateKind)
  case ukModify:
    // 1st dataset: update Fields[1], use Fields[0] in where clause
    sprintf(buffer, UpdateStmt1, DeltaDS->Fields->Fields[1]->NewValue,
            DeltaDS->Fields->Fields[0]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    // 2nd dataset: update Fields[2], use Fields[3] in where clause
    sprintf(buffer, UpdateStmt2, DeltaDS->Fields->Fields[2]->NewValue,
            DeltaDS->Fields->Fields[3]->OldValue);
    pConn->Execute(buffer, NULL, NULL);
    break;
```

```
    case ukDelete:
      // 1st dataset: use Fields[0] in where clause
      sprintf(buffer, DeleteStmt1, DeltaDS->Fields->Fields[0]->OldValue);
      pConn->Execute(buffer, NULL, NULL);
      // 2nd dataset: use Fields[3] in where clause
      sprintf(buffer, DeleteStmt2, DeltaDS->Fields->Fields[3]->OldValue);
      pConn->Execute(buffer, NULL, NULL);
      break;
    case ukInsert:
      // 1st dataset: values in Fields[0] and Fields[1]
      sprintf(buffer, UpdateStmt1, DeltaDS->Fields->Fields[0]->NewValue,
              DeltaDS->Fields->Fields[1]->NewValue);
      pConn->Execute(buffer, NULL, NULL);
      // 2nd dataset: values in Fields[2] and Fields[3]
      sprintf(buffer, UpdateStmt2, DeltaDS->Fields->Fields[2]->NewValue,
              DeltaDS->Fields->Fields[3]->NewValue);
      pConn->Execute(buffer, NULL, NULL);
      break;
  }
```

See Also

Reconciling Update Errors (see page 1737)

Applying Updates (see page 1711)

3.2.3.8.16 Limiting Records with Parameters

If the client dataset is

- a *TClientDataSet* instance whose associated provider represents a *TTable* or *TSQLTable* component
- a *TSimpleDataSet or a TBDEClientDataSet* instance whose *CommandType* property is *ctTable*

then it can use the Params property to limit the records that it caches in memory. Each parameter represents a field value that must be matched before a record can be included in the client dataset's data. This works much like a filter, except that with a filter, the records are still cached in memory, but unavailable.

Each parameter name must match the name of a field. When using *TClientDataSet*, these are the names of fields in the *TTable* or *TSQLTable* component associated with the provider. When using *TSimpleDataSet* or *TBDEClientDataSet*, these are the names of fields in the table on the database server. The data in the client dataset then includes only those records whose values on the corresponding fields match the values assigned to the parameters.

For example, consider an application that displays the orders for a single customer. When the user identifies the customer, the client dataset sets its *Params* property to include a single parameter named CustID (or whatever field in the source table is called) whose value identifies the customer whose orders should be displayed. When the client dataset requests data from the source dataset (see page 1725), it passes this parameter value. The provider then sends only the records for the identified customer. This is more efficient than letting the provider send all the orders records to the client application and then filtering the records using the client dataset.

See Also

Sending Query or Stored Procedure Parameters (see page 1727)

Limiting What Records Appear (see page 1741)

3.2.3.8.17 Merging Changes into Data

When you edit the data in a client dataset, all edits to the data exist only in an in-memory change log. This log can be maintained separately from the data itself, although it is completely transparent to objects that use the client dataset. That is, controls that navigate the client dataset or display its data see a view of the data that includes the changes. If you do not want to back out of changes, however, you should merge the change log into the data of the client dataset by calling the MergeChangeLog method. *MergeChangeLog* overwrites records in *Data* with any changed field values in the change log.

After *MergeChangeLog* executes, Data contains a mix of existing data and any changes that were in the change log. This mix becomes the new *Data* baseline against which further changes can be made. *MergeChangeLog* clears the change log of all records and resets the ChangeCount property to *0*.

Warning: Do not call *MergeChangeLog* for client datasets that use a provider. In this case, call *ApplyUpdates* to write changes to the database. For more information, see Applying updates (see page 1711).

Note: It is also possible to merge changes into the data of a separate client dataset if that dataset originally provided the data in the *Data* property. To do this, you must use a dataset provider. For an example of how to do this, see Assigning data directly (see page 1738).

If you do not want to use the extended undo capabilities of the change log, you can set the client dataset's LogChanges property to *False*. When *LogChanges* is *False*, edits are automatically merged when you post records and there is no need to call *MergeChangeLog*.

See Also

Creating a New Dataset (see page 1714)

Loading Data from a File or Stream (see page 1739)

Saving Data to a File or Stream (see page 1740)

Editing Data (see page 1715)

3.2.3.8.18 Navigating Data in Client Datasets

If an application uses standard data-aware controls, then a user can navigate through a client dataset's records using the built-in behavior of those controls. You can also navigate programmatically through records using standard dataset methods such as *First*, *Last*, *Next*, and *Prior*. For more information about these methods, see Navigating datasets (see page 1634).

Unlike most datasets, client datasets can also position the cursor at a specific record in the dataset by using the RecNo property. Ordinarily an application uses *RecNo* to determine the record number of the current record. Client datasets can, however, set *RecNo* to a particular record number to make that record the current one.

See Also

Limiting what records appear (see page 1741)

Constraining Data Values (see page 1713)

Editing Data (see page 1715)

Sorting and Indexing (see page 1728)

Representing Calculated Values (see page 1725)

Adding Application-specific Information to the Data (see page 1709)

Copying Data from Another Dataset (see page 1714)

3.2.3.8.19 Obtaining Aggregate Values

To get the value of a maintained aggregate, call the Value method of the *TAggregate* object that represents the aggregate. *Value* returns the maintained aggregate for the group that contains the current record of the client dataset.

When you are summarizing over the entire client dataset, you can call *Value* at any time to obtain the maintained aggregate. However, when you are summarizing over grouped information, you must be careful to ensure that the current record is in the

group whose summary you want. Because of this, it is a good idea to obtain aggregate values at clearly specified times, such as when you move to the first record of a group or when you move to the last record of a group. Use the GetGroupState method to determine where the current record falls within a group.

To display maintained aggregates in data-aware controls, use the Fields editor to create a persistent aggregate field component. When you specify an aggregate field in the Fields editor, the client dataset's Aggregates is automatically updated to include the appropriate aggregate specification. The AggFields property contains the new aggregated field component, and the FindField method returns it.

See Also

Specifying Aggregates (see page 1730)

Aggregating Over Groups of Records (see page 1710)

3.2.3.8.20 Overriding the Dataset On the Application Server

The client datasets that are associated with a particular data access mechanism use the CommandText and *Com*mandType properties to specify the data they represent. When using *TClientDataSet*, however, the data is specified by the source dataset, not the client dataset. Typically, this source dataset has a property that specifies an SQL statement to generate the data or the name of a database table or stored procedure.

If the provider allows, *TClientDataSet* can override the property on the source dataset that indicates what data it represents. That is, if the provider permits, the client dataset's CommandText property replaces the property on the provider's dataset that specifies what data it represents. This allows *TClientDataSet* to specify dynamically what data it wants to see.

By default, external provider components do not let client datasets use the *CommandText* value in this way. To allow *TClientDataSet* to use its *CommandText* property, you must add *poAllowCommandText* to the Options property of the provider. Otherwise, the value of *CommandText* is ignored.

Note: Never remove *poAllowCommandText* from the *Options* property of *TBDEClientDataSet* or *TIBClientDataSet*. The client dataset's *Options* property is forwarded to the internal provider, so removing *poAllowCommandText* prevents the client dataset from specifying what data to access.

The client dataset sends its *CommandText* string to the provider at two times:

- When the client dataset first opens. After it has retrieved the first data packet from the provider, the client dataset does not send *CommandText* when fetching subsequent data packets.
- When the client dataset sends an Execute command to provider.

To send an SQL command or to change a table or stored procedure name at any other time, you must explicitly use the *IAppServer* interface that is available as the AppServer property. This property represents the interface through which the client dataset communicates with its provider.

See Also

Specifying a Provider (see page 1729)

Requesting Data from the Source Dataset or Document (see page 1725)

Passing Parameters to the Application Server (see page 1723)

Getting Parameters From the Application Server (see page 1715)

Handling Constraints from the Server (see page 1716)

Refreshing Records (see page 1724)

Communicating with Providers Using Custom Events (see page 1712)

3.2.3.8.21 Overview of Using Cached Updates

To use cached updates, the following order of processes must occur in an application:

Indicate the data you want to edit. How you do this depends on the type of client dataset you are using:

- If you are using *TClientDataSet*, Specify the provider component (see page 1729) that represent the data you want to edit.
- If you are using a client dataset associated with a particular data access mechanism, you must
- Identify the database server by setting the *DBConnection* property to an appropriate connection component.
- Indicate what data you want to see by specifying the CommandText and *CommandType* properties. *CommandType* indicates whether *CommandText* is an SQL statement to execute, the name of a stored procedure, or the name of a table. If *CommandText* is a query or stored procedure, use the Params property to provide any input parameters.
- Optionally, use the Options property to indicate whether nested detail sets and BLOB data should be included in data packets or fetched separately, whether specific types of edits (insertions, modifications, or deletions) should be disabled, whether a single update can affect multiple server records, and whether the client dataset's records are refreshed when it applies updates. *Options* is identical to a provider's *Options* property. As a result, it allows you to set options that are not relevant or appropriate. For example, there is no reason to include *poIncFieldProps*, because the client dataset does not fetch its data from a dataset with persistent fields. Conversely, you do not want to exclude *poAllowCommandText*, which is included by default, because that would disable the *CommandText* property, which the client dataset uses to specify what data it wants. For information on the provider's *Options* property, see Setting options that influence the data packets (see page 1818).

Display and edit the data, permit insertion of new records, and support deletions of existing records. Both the original copy of each record and any edits to it are stored in memory.This process is described in Editing data (see page 1715).

Fetch additional records as necessary. By default, client datasets fetch all records and store them in memory. If a dataset contains many records or records with large BLOB fields, you may want to change this so that the client dataset fetches only enough records for display and re-fetches as needed. For details on how to control the record-fetching process, see Requesting data from the source dataset or document (see page 1725).

Optionally, refresh the records. As time passes, other users may modify the data on the database server. This can cause the client dataset's data to deviate more and more from the data on the server, increasing the chance of errors when you apply updates. To mitigate this problem, you can refresh records that have not already been edited. See Refreshing records (see page 1724) for details.

Apply the locally cached records to the database or cancel the updates. For each record written to the database, a *BeforeUpdateRecord* event is triggered. If an error occurs when writing an individual record to the database, an *OnUpdateError* event enables the application to correct the error, if possible, and continue updating. When updates are complete, all successfully applied updates are cleared from the local cache. For more information about applying updates to the database, see Updating records (see page 1731).

Instead of applying updates, an application can cancel the updates, emptying the change log without writing the changes to the database. You can cancel the updates by calling CancelUpdates method. All deleted records in the cache are undeleted, modified records revert to original values, and newly inserted record simply disappear.

See Also

Choosing the Type of Dataset for Caching Updates (see page 1711)

Updating Records (see page 1731)

Indicating What Records Are Modified (see page 1717)

3.2.3.8.22 Passing Parameters to the Source Dataset

Client datasets can pass parameters to the source dataset to specify what data they want provided in the data packets it sends. These parameters can specify

- Input parameter values for a query or stored procedure that is run on the application server (see page 1727)
- Field values that limit the records sent in data packets (see page 1720)

You can specify parameter values that your client dataset sends to the source dataset at design time or at runtime. At design time, select the client dataset and double-click the Params property in the **Object Inspector**. This brings up the collection editor, where you can add, delete, or rearrange parameters. By selecting a parameter in the collection editor, you can use the **Object Inspector** to edit the properties of that parameter.

At runtime, use the CreateParam method of the *Params* property to add parameters to your client dataset. *CreateParam* returns a parameter object, given a specified name, parameter type, and datatype. You can then use the properties of that parameter object to assign a value to the parameter.

For example, the following code adds an input parameter named CustNo with a value of 605:

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
AsInteger := 605;
TParam *pParam = ClientDataSet1->Params->CreateParam(ftInteger, "CustNo", ptInput);
pParam->AsInteger = 605;
```

If the client dataset is not active, you can send the parameters to the application server and retrieve a data packet that reflects those parameter values simply by setting the *Active* property to *True*.

Note: You may want to initialize parameter values from the current settings on the source dataset. You can do this by right-clicking the client dataset and choosing Fetch Params at design time or calling the FetchParams method at runtime.

See Also

Specifying a Provider (see page 1729)

Requesting Data from the Source Dataset or Document (see page 1725)

Getting Parameters From the Application Server (see page 1715)

Handling Constraints from the Server (see page 1716)

Refreshing Records (see page 1724)

Communicating with Providers Using Custom Events (see page 1712)

Overriding the Dataset On the Application Server (see page 1722)

3.2.3.8.23 Refreshing Records

Client datasets work with an in-memory snapshot of the data from the source dataset. If the source dataset represents server data, then as time elapses other users may modify that data. The data in the client dataset becomes a less accurate picture of the underlying data.

Like any other dataset, client datasets have a Refresh method that updates its records to match the current values on the server. However, calling *Refresh* only works if there are no edits in the change log. Calling *Refresh* when there are unapplied edits results in an exception.

Client datasets can also update the data while leaving the change log intact. To do this, call the RefreshRecord method. Unlike the *Refresh* method, *RefreshRecord* updates only the current record in the client dataset. *RefreshRecord* changes the record value originally obtained from the provider but leaves any changes in the change log.

Warning: It is not always appropriate to call *RefreshRecord*. If the user's edits conflict with changes made to the underlying dataset by other users, calling *RefreshRecord* masks this conflict. When the client dataset applies its updates (see page 1711), no reconcile error occurs and the application can't resolve the conflict.

In order to avoid masking update errors, you may want to check that there are no pending updates before calling *RefreshRecord*. For example, the following *AfterScroll* refreshes the current record every time the user moves to a new record (ensuring the most up-to-date value), but only when it is safe to do so.:

```
procedure TForm1.ClientDataSet1AfterScroll(DataSet: TDataSet);
begin
if ClientDataSet1.UpdateStatus = usUnModified then
```

```
ClientDataSet1.RefreshRecord;
end;
void __fastcall TForm1::ClientDataSet1AfterScroll(TDataSet *DataSet)
{
  if (ClientDataSet1->UpdateStatus == usUnModified)
    ClientDataSet1->RefreshRecord();
}
```

See Also

Specifying a Provider (see page 1729)

Requesting Data from the Source Dataset or Document (see page 1725)

Passing Parameters to the Application Server (see page 1723)

Getting Parameters From the Application Server (see page 1715)

Handling Constraints from the Server (see page 1716)

Communicating with Providers Using Custom Events (see page 1712)

Overriding the Dataset On the Application Server (see page 1722)

3.2.3.8.24 Representing Calculated Values

As with any dataset, you can add calculated fields (see page 1872) to your client dataset. These are fields whose values you calculate dynamically, usually based on the values of other fields in the same record.

Client datasets, however, let you optimize when fields are calculated by using internally calculated fields (see page 1735).

You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates (see page 1735).

See Also

Navigating Data in Client Datasets (see page 1721)

Limiting What Records Appear (see page 1741)

Constraining Data Values (see page 1713)

Editing Data (see page 1715)

Sorting and Indexing (see page 1728)

Adding Application-specific Information to the Data (see page 1709)

Copying Data from Another Dataset (see page 1714)

3.2.3.8.25 Requesting Data from the Source Dataset or Document

Client datasets can control how they fetch their data packets from a provider. By default, they retrieve all records from the source dataset. This is true whether the source dataset and provider are internal components (as with *TBDEClientDataSet,TSimpleDataSet,* and *TIBClientDataSet*), or separate components that supply the data for *TClientDataSet*.

You can change how the client dataset fetches records using the *PacketRecords* and *FetchOnDemand* properties.

Incremental fetching

By changing the PacketRecords property, you can specify that the client dataset fetches data in smaller chunks. *PacketRecords* specifies either how many records to fetch at a time, or the type of records to return. By default, *PacketRecords* is set to *-1*, which means that all available records are fetched at once, either when the client dataset is first opened, or when the application

explicitly calls *GetNextPacket*. When *PacketRecords* is *-1*, then after the client dataset first fetches data, it never needs to fetch more data because it already has all available records.

To fetch records in small batches, set *PacketRecords* to the number of records to fetch. For example, the following statement sets the size of each data packet to ten records:

```
ClientDataSet1.PacketRecords := 10;
ClientDataSet1->PacketRecords = 10;
```

This process of fetching records in batches is called "incremental fetching". Client datasets use incremental fetching when *PacketRecords* is greater than zero.

To fetch each batch of records, the client dataset calls GetNextPacket. Newly fetched packets are appended to the end of the data already in the client dataset. *GetNextPacket* returns the number of records it fetches. If the return value is the same as *PacketRecords*, the end of available records was not encountered. If the return value is greater than *0* but less than *PacketRecords*, the last record was reached during the fetch operation. If *GetNextPacket* returns *0*, then there are no more records to fetch.

Warning: Incremental fetching does not work if you are fetching data from a remote provider on a stateless application server. See Supporting state information in remote data modules (see page 1540) for information on how to use incremental fetching with stateless remote data modules.

Note: You can also use *PacketRecords* to fetch metadata information about the source dataset. To retrieve metadata information, set *PacketRecords* to *0*.

Fetch-on-demand

Automatic fetching of records is controlled by the *FetchOnDemand property. When FetchOnDemand is True (the default), the client dataset automatically fetches records as needed. To prevent automatic fetching of records, set FetchOnDemand to False. When FetchOnDemand is False, the application must explicitly call GetNextPacket to fetch records.*

For example, Applications that need to represent extremely large read-only datasets can turn off FetchOnDemand to ensure that the client datasets do not try to load more data than can fit into memory. Between fetches, the client dataset frees its cache using the EmptyDataSet method. This approach, however, does not work well when the client must post updates to the server.

The provider controls whether the records in data packets include BLOB data and nested detail datasets. If the provider excludes this information from records, the FetchOnDemand property causes the client dataset to automatically fetch BLOB data and detail datasets on an as-needed basis. If FetchOnDemand is False, and the provider does not include BLOB data and detail datasets with records, you must explicitly call the FetchBlobs or FetchDetails method to retrieve this information.

See Also

Specifying a Provider (see page 1729)

Passing Parameters to the Application Server (see page 1723)

Getting Parameters From the Application Server (see page 1715)

Handling Constraints from the Server (see page 1716)

Refreshing Records (see page 1724)

Communicating with Providers Using Custom Events (see page 1712)

Overriding the Dataset On the Application Server (see page 1722)

3.2.3.8.26 Saving Changes

Client datasets use different mechanisms for incorporating changes from the change log, depending on whether the client

datasets stores its data in a file or represents data obtained through a provider. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated.

File-based applications can simply merge the changes into the local cache represented by the *Data* property. They do not need to worry about resolving local edits with changes made by other users. To merge the change log into the *Data* property, call the *MergeChangeLog* method. Merging changes into data (see page 1720) describes this process.

You can't use *MergeChangeLog* if you are using the client dataset to cache updates or to represent the data from an external provider component. The information in the change log is required for resolving updated records with the data stored in the database (or source dataset). Instead, you call *ApplyUpdates*, which attempts to write the modifications to the database server or source dataset, and updates the *Data* property only when the modifications have been successfully committed. See Applying updates (see page 1711) for more information about this process.

See Also

Undoing Changes (see page 1730)

3.2.3.8.27 Sending Query or Stored Procedure Parameters

When the client dataset's *CommandType* property is *ctQuery* or *ctStoredProc*, or, if the client dataset is a *TClientDataSet* instance, when the associated provider represents the results of a query or stored procedure, you can use the Params property to specify parameter values. When the client dataset requests data from the source dataset (see page 1725) or uses its Execute method to run a query or stored procedure that does not return a dataset, it passes these parameter values along with the request for data or the execute command. When the provider receives these parameter values, it assigns them to its associated dataset. It then instructs the dataset to execute its query or stored procedure using these parameter values, and, if the client dataset requested data, begins providing data, starting with the first record in the result set.

Note: Parameter names should match the names of the corresponding parameters on the source dataset.

See Also

Limiting Records with Parameters (see page 1720)

Using Parameters in Queries (see page 1617)

Working with Stored Procedure Parameters (see page 1625)

3.2.3.8.28 Setting Up a Simple Dataset

Setting up a simple dataset requires two essential steps. Set up:

1. The connection information.
2. The dataset information.

The following steps describe setting up a simple dataset in more detail.

To use TSimpleDataSet:

1. Place the *TSimpleDataSet* component in a data module or on a form. Set its Name property to a unique value appropriate to your application.
2. Identify the database server that contains the data. There are two ways to do this:

- If you have a named connection in the connections file, expand the *Connectionproperty and specify the ConnectionName value.*
- For greater control over connection properties, transaction support, login support, and the ability to use a single connection for more than one dataset, use a separate *TSQLConnection* component instead. Specify the *TSQLConnection* component as the value of the Connection property. For details on *TSQLConnection*, see Connecting to databases (see page 1506).

3. To indicate what data you want to fetch from the server, expand the DataSet property and set the appropriate values. There are three ways to fetch data from the server:

- Set CommandType to *ctQuery* and set CommandText to an SQL statement you want to execute on the server. This statement is typically a SELECT statement. Supply the values for any parameters using the Params property.
- Set CommandType to *ctStoredProc* and set CommandText to the name of the stored procedure you want to execute. Supply the values for any input parameters using the Params property.
- Set CommandType to *ctTable* and set CommandText to the name of the database tables whose records you want to use.

4. If the data is to be used with visual data controls, add a data source component to the form or data module, and set its *DataSet* property to the *TSimpleDataSet* object. The data source component forwards the data in the client dataset's in-memory cache to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
5. Activate the dataset by setting the *Active* property to *true* (or, at runtime, calling the *Open* method).
6. If you executed a stored procedure, use the Params property to retrieve any output parameters.
7. When the user has edited the data in the simple dataset, you can apply those edits back to the database server by calling the ApplyUpdates method. Resolve any update errors in an OnReconcileError event handler. For more information on applying updates, see Updating records (see page 1731).

See Also

When to Use TSimpleDataset (see page 1736)

3.2.3.8.29 Sorting and Indexing

Using indexes provides several benefits to your applications:

- They allow client datasets to locate data quickly.
- They let you apply ranges to limit the available records.
- They let your application set up relationships with other datasets such as lookup tables or master/detail forms.
- They specify the order in which records appear.

If a client dataset represents server data or uses an external provider, it inherits a default index and sort order based on the data it receives. The default index is called DEFAULT_ORDER. You can use this ordering, but you cannot change or delete the index.

In addition to the default index, the client dataset maintains a second index, called CHANGEINDEX, on the changed records stored in the change log (*Delta* property). CHANGEINDEX orders all records in the client dataset as they would appear if the changes specified in *Delta* were applied. CHANGEINDEX is based on the ordering inherited from DEFAULT_ORDER. As with DEFAULT_ORDER, you cannot change or delete the CHANGEINDEX index.

You can use other existing indexes, and you can create your own indexes. The following sections describe how to create and use indexes with client datasets:

- Adding a new index (see page 1709)
- Deleting and switching indexes (see page 1715)
- Using indexes to group data (see page 1734)

Note: You may also want to review the material on indexes in table type datasets, which also applies to client datasets. This material is in Using Indexes to search for records (see page 1615) and Limiting records with ranges (see page 1597).

See Also

Navigating Data in Client Datasets (see page 1721)

Limiting What Records Appear (see page 1741)

Constraining Data Values (see page 1713)

Editing Data (see page 1715)

Representing Calculated Values (see page 1725)

Adding Application-specific Information to the Data (see page 1709)

Copying Data from Another Dataset (see page 1714)

3.2.3.8.30 Specifying a Provider

Unlike the client datasets that are associated with a data access mechanism, *TClientDataSet* has no internal provider component to package data or apply updates. If you want it to represent data from a source dataset or XML document, therefore, you must associated the client dataset with an external provider component.

The way you associate *TClientDataSet* with a provider depends on whether the provider is in the same application as the client dataset or on a remote application server running on another system.

Provider's location	How to associate TClientDataSet
The provider is in the same application as the client dataset	If the provider is in the same application as the client dataset, you can associate it with a provider by choosing a provider from the drop-down list for the ProviderName property in the **Object Inspector**. This works as long as the provider has the same Owner as the client dataset. (The client dataset and the provider have the same *Owner* if they are placed in the same form or data module.) To use a local provider that has a different *Owner*, you must form the association at runtime using the client dataset's SetProvider method If you think you may eventually scale up to a remote provider, or if you want to make calls directly to the *IAppServer* interface, you can also set the ConnectionBroker property to a TLocalConnection component. If you use *TLocalConnection*, the *TLocalConnection* instance manages the list of all providers that are local to the application, and handles the client dataset's *IAppServer* calls. If you do not use *TLocalConnection*, the application creates a hidden object that handles the *IAppServer* calls from the client dataset.
The provider is on a remote application server	If the provider is on a remote application server, then, in addition to the ProviderName property, you need to specify a component that connects the client dataset to the application server. There are two properties that can handle this task: RemoteServer, which specifies the name of a connection component from which to get a list of providers, or *ConnectionBroker, which specifies a centralized broker that provides an additional level of indirection between the client dataset and the connection component. The connection component and, if used, the connection broker, reside in the same data module as the client dataset. The connection component establishes and maintains a connection to an application server, sometimes called a "data broker." For more information, see The structure of the client application (see page 1543)* *At design time, after you specify RemoteServer or ConnectionBroker, you can select a provider from the drop-down list for the ProviderName property in the **Object Inspector**. This list includes both local providers (in the same form or data module) and remote providers that can be accessed through the connection component.*

Note: If the connection component is an instance of *TDCOMConnection*, the application server must be registered on the client machine.

At runtime, you can switch among available providers (both local and remote) by setting *ProviderName* in code.

See Also

Requesting Data from the Source Dataset or Document (see page 1725)

Passing Parameters to the Application Server (see page 1723)

Getting Parameters From the Application Server (see page 1715)

Handling Constraints from the Server (see page 1716)

Refreshing Records (see page 1724)

Communicating with Providers Using Custom Events (see page 1712)

Overriding the Dataset on the Application Server (see page 1722)

3.2.3.8.31 Specifying Aggregates

To specify that you want to calculate summaries over the records in a client dataset, use the Aggregates property. *Aggregates* is a collection of aggregate specifications (TAggregate). You can add aggregate specifications to your client dataset using the Collection Editor at design time, or using the Add method of *Aggregates* at runtime. If you want to create field components for the aggregates, create persistent fields for the aggregated values in the Fields Editor.

Note: When you create aggregated fields, the appropriate aggregate objects are added to the client dataset's *Aggregates* property automatically. Do not add them explicitly when creating aggregated persistent fields (see page 1856).

For each aggregate, the Expression property indicates the summary calculation it represents. *Expression* can contain a simple summary expression such as

```
Sum(Field1)
```

or a complex expression that combines information from several fields, such as

```
Sum(Qty * Price) - Sum(AmountPaid)
```

Aggregate expressions include one or more of the summary operators in the following table

Summary operators for maintained aggregates

Operator	Use
Sum	Totals the values for a numeric field or expression
Avg	Computes the average value for a numeric or date-time field or expression
Count	Specifies the number of non-blank values for a field or expression
Min	Indicates the minimum value for a string, numeric, or date-time field or expression
Max	Indicates the maximum value for a string, numeric, or date-time field or expression

The summary operators act on field values or on expressions built from field values using the same operators you use to create filters. (You can't nest summary operators, however.) You can create expressions by using operators on summarized values with other summarized values, or on summarized values and constants. However, you can't combine summarized values with field values, because such expressions are ambiguous (there is no indication of which record should supply the field value.) These rules are illustrated in the following expressions:

Sum(Qty * Price)	{legal -- summary of an expression on fields }
Max(Field1) - Max(Field2)	{legal -- expression on summaries }
Avg(DiscountRate) * 100	{legal -- expression of summary and constant }
Min(Sum(Field1))	{illegal -- nested summaries }
Count(Field1) - Field2	{illegal -- expression of summary and field }

See Also

Aggregating Over Groups of Records (see page 1710)

Obtaining Aggregate Values (see page 1721)

3.2.3.8.32 Undoing Changes

Even though a record's original version remains unchanged in *Data*, each time a user edits a record, leaves it, and returns to it, the user sees the last changed version of the record. If a user or application edits a record a number of times, each changed version of the record is stored in the change log as a separate entry.

Storing each change to a record makes it possible to support multiple levels of undo operations should it be necessary to restore a record's previous state:

- To remove the last change to a record, call UndoLastChange. *UndoLastChange* takes a Boolean parameter, *FollowChange*, that indicates whether to reposition the cursor on the restored record (*True*), or to leave the cursor on the current record (*False*). If there are several changes to a record, each call to *UndoLastChange* removes another layer of edits. *UndoLastChange* returns a Boolean value indicating success or failure. If the removal occurs, *UndoLastChange* returns *True*. Use the ChangeCount property to check whether there are more changes to undo. *ChangeCount* indicates the number of changes stored in the change log.
- Instead of removing each layer of changes to a single record, you can remove them all at once. To remove all changes to a record, select the record, and call RevertRecord. *RevertRecord* removes any changes to the current record from the change log.
- To restore a deleted record, first set the StatusFilter property to [*usDeleted*], which makes the deleted records "visible." Next, navigate to the record you want to restore and call *RevertRecord*. Finally, restore the *StatusFilter* property to [*usModified*, *usInserted*, *usUnmodified*] so that the edited version of the dataset (now containing the restored record) is again visible.
- At any point during edits, you can save the current state of the change log using the SavePoint property. Reading *SavePoint* returns a marker into the current position in the change log. Later, if you want to undo all changes that occurred since you read the save point, set *SavePoint* to the value you read previously. Your application can obtain values for multiple save points. However, once you back up the change log to a save point, the values of all save points that your application read after that one are invalid.
- You can abandon all changes recorded in the change log by calling CancelUpdates. *CancelUpdates* clears the change log, effectively discarding all edits to all records. Be careful when you call *CancelUpdates*. After you call *CancelUpdates*, you cannot recover any changes that were in the log.

See Also

Saving Changes (see page 1726)

3.2.3.8.33 Updating Records

The contents of the change log are stored as a data packet in the client dataset's *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database (or source dataset or XML document).

When a client applies updates to the server, the following steps occur:

1. The client application calls the ApplyUpdates method (see page 1711) of a client dataset object. This method passes the contents of the client dataset's *Delta* property to the (internal or external) provider. *Delta* is a data packet that contains a client dataset's updated, inserted, and deleted records.
2. The provider applies the updates, caching any problem records that it can't resolve itself. See Responding to client update requests (see page 1808) for details on how the provider applies updates.
3. The provider returns all unresolved records to the client dataset in a *Result* data packet. The *Result* data packet contains all records that were not updated. It also contains error information, such as error messages and error codes.
4. The client dataset attempts to reconcile update errors (see page 1737) returned in the *Result* data packet on a record-by-record basis.

See Also

Overview of Using Cached Updates (see page 1723)

Indicating What Records Are Modified (see page 1717)

Choosing the Type of Dataset for Caching Updates (see page 1711)

3.2.3.8.34 Using a Client Dataset to Cache Updates

By default, when you edit data in most datasets, every time you delete or post a record, the dataset generates a transaction, deletes or writes that record to the database server, and commits the transaction. If there is a problem writing changes to the

database, your application is notified immediately: the dataset raises an exception when you post the record.

If your dataset uses a remote database server, this approach can degrade performance due to network traffic between your application and the server every time you move to a new record after editing the current record. To minimize the network traffic, you may want to cache updates locally. When you cache updates, you application retrieves data from the database, caches and edits it locally, and then applies the cached updates to the database in a single transaction. When you cache updates, changes to a dataset (such as posting changes or deleting records) are stored locally instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

Caching updates can minimize transaction times and reduce network traffic. However, cached data is local to your application and is not under transaction control. This means that while you are working on your local, in-memory, copy of the data, other applications can be changing the data in the underlying database table. They also can't see any changes you make until you apply the cached updates. Because of this, cached updates may not be appropriate for applications that work with volatile data, as you may create or encounter too many conflicts when trying to merge your changes into the database.

Although the BDE and ADO provide alternate mechanisms for caching updates, using a client dataset for caching updates has several advantages:

- Applying updates when datasets are linked in master/detail relationships is handled for you. This ensures that updates to multiple linked datasets are applied in the correct order.
- Client datasets give you the maximum of control over the update process. You can set properties to influence the SQL that is generated for updating records, specify the table to use when updating records from a multi-table join, or even apply updates manually from a *BeforeUpdateRecord* event handler.
- When errors occur applying cached updates to the database server, only client datasets (and dataset providers) provide you with information about the current record value on the database server in addition to the original (unedited) value from your dataset and the new (edited) value of the update that failed.
- Client datasets let you specify the number of update errors you want to tolerate before the entire update is rolled back.

The following topics describe in more detail on how to use a client dataset to cache updates:

- Overview of using cached updates (see page 1723).
- Choosing the type of dataset for caching updates (see page 1711).
- Indicating what records are modified (see page 1717).
- Updating records (see page 1731).

See Also

Working with Data Using a Client Dataset (see page 1737)

Using a Client Dataset with a Provider (see page 1732)

Using a Client Dataset with Flat File Data (see page 1733)

3.2.3.8.35 Using a Client Dataset with a Provider

A client dataset uses a provider to supply it with data and apply updates when

- It caches updates from a database server or another dataset.
- It represents the data in an XML document.
- It stores the data in the client portion of a multi-tiered application.

For any client dataset other than *TClientDataSet*, this provider is internal, and so not directly accessible by the application. With *TClientDataSet*, the provider is an external component that links the client dataset to an external source of data.

An external provider component can reside in the same application as the client dataset, or it can be part of a separate application running on another system. For more information about provider components, see Using Provider Components (

see page 1819). For more information about applications where the provider is in a separate application on another system, see Creating multi-tiered applications (see page 1518).

When using an (internal or external) provider, the client dataset always caches any updates. For information on how this works, see Using a client dataset to cache updates (see page 1731).

The following topics describe additional properties and methods of the client dataset that enable it to work with a provider:

- Specifying a provider (see page 1729)
- Requesting data from the source dataset or document (see page 1725).
- Getting Parameters From the Application Server (see page 1715)
- Passing parameters to the source dataset (see page 1723)
- Handling constraints from the server (see page 1716)
- Refreshing records (see page 1724).
- Communicating with providers using custom events (see page 1712)
- Overriding the source dataset (see page 1722)

See Also

Working with Data Using a Client Dataset (see page 1737)

Using a Client Dataset to Cache Updates (see page 1731)

Using a Client Dataset with Flat File Data (see page 1733)

Creating Multi-tiered Applications (see page 1518)

3.2.3.8.36 Using a Client Dataset with File-based Data

Client datasets can work with dedicated files on disk as well as server data. This allows them to be used in file-based database applications (see page 1564) and "briefcase model" applications (see page 1565). The special files that client datasets use for their data are called MyBase.

Tip: All client datasets are appropriate for a briefcase model application, but for a pure MyBase application (one that does not use a provider), it is preferable to use *TClientDataSet*, because it involves less overhead.

In a pure MyBase application, the client application cannot get table definitions and data from the server, and there is no server to which it can apply updates. Instead, the client dataset must independently

- Define and create tables (see page 1714)
- Load saved data (see page 1739)
- Merge edits into its data (see page 1720)
- Save data (see page 1740)

See Also

Working with Data Using a Client Dataset (see page 1737)

Using a Client Dataset to Cache Updates (see page 1731)

Using a Client Dataset with a Provider (see page 1732)

3.2.3.8.37 Using a Simple Dataset

TSimpleDataSet is a special type of client dataset designed for simple two-tiered applications. Like a unidirectional dataset, it can use an SQL connection component to connect to a database server and specify an SQL statement to execute on that

server. Like other client datasets, it buffers data in memory to allow full navigation and editing support.

TSimpleDataSet works the same way as a generic client dataset (*TClientDataSet*) that is linked to a unidirectional dataset by a dataset provider. In fact, *TSimpleDataSet* has its own, internal provider, which it uses to communicate with an internally created unidirectional dataset.

Using a simple dataset can simplify the process of two-tiered application development because you don't need to work with as many components.

When to use TSimpleDataSet (see page 1736) provides information on when and how to use a simple dataset:

See Also

Copying Data from Another Dataset (see page 1714)

Using a Client Dataset with File-based Data (see page 1733)

Using a Client Dataset with a Provider (see page 1732)

Using a Client Dataset to Buffer Records (see page 1559)

3.2.3.8.38 Using Indexes to Group Data

When you use an index in your client dataset, it automatically imposes a sort order on the records. Because of this order, adjacent records usually contain duplicate values on the fields that make up the index. For example, consider the following fragment from an orders table that is indexed on the SalesRep and Customer fields:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Because of the sort order, adjacent values in the SalesRep column are duplicated. Within the records for SalesRep 1, adjacent values in the Customer column are duplicated. That is, the data is grouped by SalesRep, and within the SalesRep group it is grouped by Customer. Each grouping has an associated level. In this case, the SalesRep group has level 1 (because it is not nested in any other groups) and the Customer group has level 2 (because it is nested in the group with level 1). Grouping level corresponds to the order of fields in the index.

Client datasets let you determine where the current record lies within any given grouping level. This allows your application to display records differently, depending on whether they are the first record in the group, in the middle of a group, or the last record in a group. For example, you might want to display a field value only if it is on the first record of the group, eliminating the duplicate values. To do this with the previous table results in the following:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10

	3	4	200

To determine where the current record falls within any group, use the GetGroupState method. *GetGroupState* takes an integer giving the level of the group and returns a value indicating where the current record falls the group (first record, last record, or neither).

When you create an index, you can specify the level of grouping it supports (up to the number of fields in the index). *GetGroupState* can't provide information about groups beyond that level, even if the index sorts records on additional fields.

See Also

Adding a New Index (see page 1709)

Using Maintained Aggregates (see page 1735)

Deleting and Switching Indexes (see page 1715)

3.2.3.8.39 Using Internally Calculated Fields in Client Datasets

In other datasets, your application must compute the value of calculated fields every time the record changes or the user edits any fields in the current record. It does this in an OnCalcFields event handler.

While you can still do this, client datasets let you minimize the number of times calculated fields must be recomputed by saving calculated values in the client dataset's data. When calculated values are saved with the client dataset, they must still be recomputed when the user edits the current record, but your application need not recompute values every time the current record changes. To save calculated values in the client dataset's data, use internally calculated fields instead of calculated fields.

Internally calculated fields, just like calculated fields, are calculated in an *OnC*alcFields event handler. However, you can optimize your event handler by checking the State property of your client dataset. When *State* is *dsInternalCalc*, you must recompute internally calculated fields. When *State* is *dsCalcFields*, you need only recompute regular calculated fields.

To use internally calculated fields, you must define the fields as internally calculated before you create the client dataset (see page 1587). Depending on whether you use persistent fields or field definitions, you do this in one of the following ways:

- If you use persistent fields, define fields as internally calculated by selecting InternalCalc in the Fields editor.
- If you use field definitions, set the InternalCalcField property of the relevant field definition to *True*.

  Note: Other types of datasets use internally calculated fields. However, with other datasets, you do not calculate these values in an *OnCalcFields* event handler. Instead, they are computed automatically by the BDE or remote database server.

See Also

Using Maintained Aggregates (see page 1735)

Defining a Calculated Field (see page 1872)

Programming a Calculated Field (see page 1870)

3.2.3.8.40 Using Maintained Aggregates

Client datasets provide support for summarizing data over groups of records. Because these summaries are automatically updated as you edit the data in the dataset, this summarized data is called a "maintained aggregate."

In their simplest form, maintained aggregates let you obtain information such as the sum of all values in a column of the client dataset. They are flexible enough, however, to support a variety of summary calculations and to provide subtotals over groups of records defined by a field in an index that supports grouping (see page 1734).

The following topics describe how to

- Specify aggregates (see page 1730).
- Aggregate Over Groups of Records (see page 1710).
- Obtain aggregate values (see page 1721).

See Also

Using Internally Calculated Fields in Client Datasets (see page 1735)

Using Indexes to Group Data (see page 1734)

3.2.3.8.41 When to Use TSimpleDataSet

TSimpleDataSet is intended for use in a simple two-tiered database applications and briefcase model applications. It provides an easy-to-set up component for linking to the database server, fetching data, caching updates, and applying them back to the server. It can be used in most two-tiered applications.

There are times, however, when it is more appropriate to use *TClientDataSet*:

- If you are not using data from a database server (for example, if you are using a dedicated file on disk (see page 1564)), then *TClientDataSet* has the advantage of less overhead.
- Only *TClientDataSet* can be used in a multi-tiered database application . Thus, if you are writing a multi-tiered application, or if you intend to scale up to a multi-tiered application eventually, you should use *TClientDataSet* with an external provider and source dataset.
- Because the source dataset is internal to the simple dataset component, you can't link two source datasets in a master/detail relationship to obtain nested detail sets. (You can, however, link two simple datasets into a master/detail relationship.)
- The simple dataset does not surface any of the events or properties that occur on its internal dataset provider. However, in most cases, these events are used in multi-tiered applications, and are not needed for two-tiered applications.

Setting up a simple dataset (see page 1727) provides information on setting up a simple dataset:

See Also

Setting Up a Simple Dataset (see page 1727)

3.2.3.8.42 Cloning a Client Dataset Cursor

Client datasets use the *CloneCursor* method to let you work with a second view of the data at runtime. *CloneCursor* lets a second client dataset share the original client dataset's data. This is less expensive than copying all the original data, but, because the data is shared, the second client dataset can't modify the data without affecting the original client dataset.

CloneCursor takes three parameters: *Source* specifies the client dataset to clone. The last two parameters (*Reset* and *KeepSettings*) indicate whether to copy information other than the data. This information includes any filters, the current index, links to a master table (when the source dataset is a detail set), the *ReadOnly* property, and any links to a connection component or provider.

When *Reset* and *KeepSettings* are *False*, a cloned client dataset is opened, and the settings of the source client dataset are used to set the properties of the destination. When *Reset* is *True*, the destination dataset's properties are given the default values (no index or filters, no master table, *ReadOnly* is *False*, and no connection component or provider is specified). When *KeepSettings* is *True*, the destination dataset's properties are not changed.

See Also

Assigning Data Directly (see page 1738)

3.2.3.8.43 Working with Data Using a Client Dataset

Like any dataset, you can use client datasets to supply the data for data-aware controls using a data source component. See Using data controls (see page 1778) for information on how to display database information in data-aware controls.

Client datasets implement all the properties an methods inherited from *TDataSet*. For a complete introduction to this generic dataset behavior, see Understanding datasets (see page 1632).

In addition, client datasets implement many of the features common to table type datasets (see page 1620) such as

- Sorting records with indexes (see page 1609).
- Using Indexes to search for records (see page 1615).
- Limiting records with ranges (see page 1597).
- Creating master/detail relationships (see page 1590).
- Controlling read/write access (see page 1587)
- Creating the underlying dataset (see page 1587)
- Emptying the dataset (see page 1592)
- Synchronizing client datasets (see page 1614)

Client datasets differ from other datasets in that they hold all their data in memory. Because of this, their support for some database functions can involve additional capabilities or considerations. The following topics describe some of these common functions and the differences introduced by client datasets:

- Navigating data (see page 1721)
- Limiting What Records Appear (see page 1741)
- Editing data (see page 1715) .
- Constraining data values (see page 1713)
- Sorting and indexing (see page 1728) .
- Representing calculated values (see page 1725) .
- Copying data from another dataset (see page 1714).
- Adding application-specific information to the data (see page 1709) .

See Also

Using a Client Dataset to Cache Updates (see page 1731)

Using a Client Dataset with a Provider (see page 1732)

Using a Client Dataset with Flat File Data (see page 1733)

3.2.3.8.44 Reconciling Update Errors

There are two events that let you handle errors that occur during the update process:

- During the update process, the internal provider generates an OnUpdateError event every time it encounters an update that it can't handle. If you correct the problem in an *OnUpdateError* event handler, then the error does not count toward the maximum number of errors passed to the *ApplyUpdates* method. This event only occurs for client datasets that use an internal provider. If you are using *TClientDataSet*, you can use the provider component's *OnUpdateError* event instead.
- After the entire update operation is finished, the client dataset generates an OnReconcileError event for every record that the provider could not apply to the database server.

You should always code an *OnReconcileError* or *OnUpdateError* event handler, even if only to discard the records returned that could not be applied. The event handlers for these two events work the same way. They include the following parameters:

DataSet: A client dataset that contains the updated record which couldn't be applied. You can use this dataset's methods to get information about the problem record and to edit the record in order to correct any problems. In particular, you will want to use the CurValue, OldValue, and NewValue properties of the fields in the current record to determine the cause of the update problem. However, you must not call any client dataset methods that change the current record in your event handler.

E: An object that represents the problem that occurred. You can use this exception to extract an error message or to determine the cause of the update error.

UpdateKind: The type of update that generated the error. *UpdateKind* can be *ukModify* (the problem occurred updating an existing record that was modified), *ukInsert* (the problem occurred inserting a new record), or *ukDelete* (the problem occurred deleting an existing record).

Action: A **var** parameter that indicates what action to take when the event handler exits. In your event handler, you set this parameter to

- Skip this record, leaving it in the change log. (rrSkip or raSkip)
- Stop the entire reconcile operation. (rrAbort or raAbort)
- Merge the modification that failed into the corresponding record from the server. (rrMerge or raMerge) This only works if the server record does not include any changes to fields modified in the client dataset's record.
- Replace the current update in the change log with the value of the record in the event handler, which has presumably been corrected. (rrApply or raCorrect)
- Ignore the error completely. (rrIgnore) This possibility only exists in the *OnUpdateError* event handler, and is intended for the case where the event handler applies the update back to the database server. The updated record is removed from the change log and merged into *Data*, as if the provider had applied the update.
- Back out the changes for this record on the client dataset, reverting to the originally provided values. (raCancel) This possibility only exists in the *OnReconcileError* event handler.
- Update the current record value to match the record on the server. (raRefresh) This possibility only exists in the *OnReconcileError* event handler.

The following code shows an *OnReconcileError* event handler that uses the reconcile error dialog from the RecError unit which ships in the objrepos directory. (To use this dialog, add RecError to your uses clause.)

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TCustomClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
void __fastcall TForm1::ClientDataSetReconcileError(TCustomClientDataSet *DataSet,
    EReconcileError *E, TUpdateKind UpdateKind, TReconcileAction &Action)
{
  Action = HandleReconcileError(this, DataSet, UpdateKind, E);
}
```

See Also

Applying Updates (see page 1711)

Intervening as Updates Are Applied (see page 1718)

3.2.3.8.45 Assigning Data Directly

You can use the client dataset's *Data* property to assign data to a client dataset from another dataset. *Data* is a data packet in the form of an OleVariant. A data packet can come from another client dataset or from any other dataset by using a provider. Once a data packet is assigned to *Data*, its contents are displayed automatically in data-aware controls connected to the client dataset by a data source component.

When you open a client dataset that represents server data or that uses an external provider component, data packets are automatically assigned to *Data*.

When your client dataset does not use a provider, you can copy the data from another client dataset as follows:

```
ClientDataSet1.Data := ClientDataSet2.Data;
ClientDataSet1->Data = ClientDataSet2->Data;
```

Note: When you copy the *Data* property of another client dataset, you copy the change log as well, but the copy does not reflect any filters or ranges that have been applied. To include filters or ranges, you must clone the source dataset's cursor instead.

If you are copying from a dataset other than a client dataset, you can create a dataset provider component, link it to the source dataset, and then copy its data:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = SourceDataSet;
ClientDataSet1->Data = TempProvider->Data;
delete TempProvider;
```

Note: When you assign directly to the *Data* property, the new data packet is not merged into the existing data. Instead, all previous data is replaced.

If you want to merge changes from another dataset, rather than copying its data, you must use a provider component. Create a dataset provider as in the previous example, but attach it to the destination dataset and instead of copying the data property, use the *ApplyUpdates* method:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
TempProvider = new TDataSetProvider(Form1);
TempProvider->DataSet = ClientDataSet1;
TempProvider->ApplyUpdates(SourceDataSet->Delta, -1, ErrCount);
delete TempProvider;
```

See Also

Cloning a Client Dataset Cursor (see page 1736)

Editing Data (see page 1715)

3.2.3.8.46 Loading Data from a File or Stream

To load data from a file, call a client dataset's LoadFromFile method. *LoadFromFile* takes one parameter, a string that specifies the file from which to read data. The file name can be a fully qualified path name, if appropriate. If you always load the client dataset's data from the same file, you can use the FileName property instead. If *FileName* names an existing file, the data is automatically loaded when the client dataset is opened.

To load data from a stream, call the client dataset's *LoadFromStream* method. *LoadFromStream* takes one parameter, a stream object that supplies the data.

The data loaded by *LoadFromFile* (*LoadFromStream)* must have previously been saved in a client dataset's data format by this or another client dataset using the *SaveToFile* (SaveToStream) method, or generated from an XML document. For more information about saving data to a file or stream, see Saving data to a file or stream (see page 1740). For information about creating client dataset data from an XML document, see Using XML in database applications (see page 1847).

When you call *LoadFromFile* or *LoadFromStream*, all data in the file is read into the *Data* property. Any edits that were in the change log when the data was saved are read into the *Delta* property. However, the only indexes that are read from the file are those that were created with the dataset.

See Also

Creating a New Dataset (see page 1714)

Merging Changes into Data (see page 1720)

Saving Data to a File or Stream (see page 1740)

3.2.3.8.47 Saving Data to a File or Stream

Even when you have merged changes into the data of a client dataset, this data still exists only in memory. While it persists if you close the client dataset and reopen it in your application, it will disappear when your application shuts down. To make the data permanent, it must be written to disk. Write changes to disk using the SaveToFile method.

SaveToFile takes one parameter, a string that specifies the file into which to write data. The file name can be a fully qualified path name, if appropriate. If the file already exists, its current contents are completely overwritten.

Note: *SaveToFile* does not preserve any indexes you added to the client dataset at runtime, only indexes that were added when you created the client dataset.

If you always save the data to the same file, you can use the FileName *property instead. If FileName is set, the data is automatically saved to the named file when the client dataset is closed.*

You can also save data to a stream, using the SaveToStream method. SaveToStream takes one parameter, a stream object that receives the data.

Note: *If you save a client dataset while there are still edits in the change log, these are not merged with the data. When you reload the data, using the LoadFromFile or LoadFromStream method, the change log will still contain the unmerged edits. This is important for applications that support the briefcase model, where those changes will eventually have to be applied to a provider component on the application server.*

See Also

Creating a New Dataset (see page 1714)

Merging Changes into Data (see page 1720)

Loading Data from a File or Stream (see page 1739)

Using the Briefcase Model (see page 1565)

Sorting and Indexing (see page 1728)

Editing Data (see page 1715)

3.2.3.8.48 Using Client Datasets: Overview

Client datasets are specialized datasets that hold all their data in memory. The support for manipulating the data they store in memory is provided by midaslib.dcu or midas.dll. The format client datasets use for storing data is self-contained and easily transported, which allows client datasets to

- Read from and write to dedicated files on disk, acting as a file-based dataset. Properties and methods supporting this mechanism are described in Using a client dataset with file-based data (see page 1733).
- Cache updates for data from a database server. Client dataset features that support cached updates are described in Using a client dataset to cache updates (see page 1731).
- Represent the data in the client portion of a multi-tiered application. To function in this way, the client dataset must work with an external provider, as described in Using a client dataset with a provider (see page 1732). For information about multi-tiered database applications, see Creating multi-tiered applications (see page 1518).
- Represent the data from a source other than a dataset. Because a client dataset can use the data from an external provider, specialized providers can adapt a variety of information sources to work with client datasets. For example, you can use an XML provider to enable a client dataset to represent the information in an XML document.

Whether you use client datasets for file-based data, caching updates, data from an external provider (such as working with an XML document or in a multi-tiered application), or a combination of these approaches such as a "briefcase model" application, you can take advantage of broad range of features client datasets support for working with data (see page 1737).

See Also

Designing Database Applications (see page 1566)

Using Data Controls (see page 1778)

Working with Field Components (see page 1877)

Understanding Datasets (see page 1632)

Connecting to databases (see page 1506)

Working with ADO Components (see page 1494)

Using Unidirectional Datasets (see page 1823)

Using the Borland Database Engine (see page 1643)

Using Provider Components (see page 1819)

Creating Multi-tiered Applications: Overview (see page 1518)

Using Decision Support Components (see page 1800)

Using Data Controls (see page 1778)

Using XML in database applications (see page 1847)

3.2.3.8.49 Limiting What Records Appear

To restrict users to a subset of available data on a temporary basis, applications can use ranges and filters. When you apply a range or a filter, the client dataset does not display all the data in its in-memory cache. Instead, it only displays the data that meets the range or filter conditions. For more information about using filters, see Displaying and editing a subset of data using filters (see page 1631). For more information about ranges, see Limiting records with ranges (see page 1597).

With most datasets, filter strings are parsed into SQL commands that are then implemented on the database server. Because of this, the SQL dialect of the server limits what operations are used in filter strings. Client datasets implement their own filter support, which includes more operations than that of other datasets. For example, when using a client dataset, filter expressions can include string operators that return substrings, operators that parse date/time values, and much more. Client datasets also allow filters on BLOB fields or complex field types such as ADT fields and array fields.

The various operators and functions that client datasets can use in filters, along with a comparison to other datasets that support filters, is given below:

Filter support in client datasets

Operator or function	Example	Supported by other datasets	Comment
Comparisons			
=	State = 'CA'	Yes	
<>	State <> 'CA'	Yes	
>=	DateEntered >= '1/1/1998'	Yes	
<=	Total <= 100,000	Yes	
>	Percentile > 50	Yes	

<	Field1 < Field2	Yes	
BLANK	State <> 'CA' or State = BLANK	Yes	Blank records do not appear unless explicitly included in the filter.
IS NULL	Field1 IS NULL	No	
IS NOT NULL	Field1 IS NOT NULL	No	
Logical operators			
and	State = 'CA' and Country = 'US'	Yes	
or	State = 'CA' or State = 'MA'	Yes	
not	not (State = 'CA')	Yes	
Arithmetic operators			
+	Total + 5 > 100	Depends on driver	Applies to numbers, strings, or date (time) + number.
-	Field1 - 7 <> 10	Depends on driver	Applies to numbers, dates, or date (time) - number.
*	Discount * 100 > 20	Depends on driver	Applies to numbers only.
/	Discount > Total / 5	Depends on driver	Applies to numbers only.
String functions			
Upper	Upper(Field1) = 'ALWAYS'	No	
Lower	Lower(Field1 + Field2) = 'josp'	No	
Substring	Substring(DateFld,8) = '1998' Substring(DateFld,1,3) = 'JAN'	No	Value goes from position of second argument to end or number of chars in third argument. First char has position 1.
Trim	Trim(Field1 + Field2) Trim(Field1, '-')	No	Removes third argument from front and back. If no third argument, trims spaces.
TrimLeft	TrimLeft(StringField) TrimLeft(Field1, '$') <> ''	No	See Trim.
TrimRight	TrimRight(StringField) TrimRight(Field1, '.') <> ''	No	See Trim.
DateTime functions			
Year	Year(DateField) = 2000	No	
Month	Month(DateField) <> 12	No	
Day	Day(DateField) = 1	No	
Hour	Hour(DateField) < 16	No	
Minute	Minute(DateField) = 0	No	
Second	Second(DateField) = 30	No	
GetDate	GetDate - DateField > 7	No	Represents current date and time.
Date	DateField = Date(GetDate)	No	Returns the date portion of a datetime value.

Time	TimeField > Time(GetDate)	No	Returns the time portion of a datetime value.
Miscellaneous			
Like	Memo LIKE '%filters%'	No	Works like SQL-92 without the ESC clause. When applied to BLOB fields, FilterOptions determines whether case is considered.
In	Day(DateField) in (1,7)	No	Works like SQL-92. Second argument is a list of values all with the same type.
*	State = 'M*'	Yes	Wildcard for partial comparisons.

When applying ranges or filters, the client dataset still stores all of its records in memory. The range or filter merely determines which records are available to controls that navigate or display data from the client dataset.

Note: When fetching data from a provider, you can also limit the data that the client dataset stores by supplying parameters to the provider. For details, see Limiting Records with Parameters (see page 1720).

See Also

Navigating Data in Client Datasets (see page 1721)

Constraining Data Values (see page 1713)

Editing Data (see page 1715)

Sorting and Indexing (see page 1728)

Representing Calculated Values (see page 1725)

Adding Application-specific Information to the Data (see page 1709)

Copying Data from Another Dataset (see page 1714)

3.2.3.9 Using data controls

Topics

Name	Description
Associating a Data Control with a Dataset (see page 1746)	Data controls connect to datasets by using a data source. A data source component (TDataSource) acts as a conduit between the control and a dataset containing data. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form. **Note:** Data source components are also required for linking unnested datasets in master-detail relationships.
Choosing How to Organize the Data (see page 1747)	When you build the user interface for your database application, you have choices to make about how you want to organize the display of information and the controls that manipulate that information. One of the first decisions to make is whether you want to display a single record at a time, or multiple records (see page 1760). In addition, you will want to add controls to navigate and manipulate records (see page 1767). The TDBNavigator control provides built-in support for many of the functions you may want to perform.
Changing the Associated Dataset at Runtime (see page 1747)	In Associating a Data Control with a Dataset (see page 1746), the datasource was associated with its dataset by setting the *DataSet* property at design time. At runtime, you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the *CustSource* data source component between the dataset components named *Customers* and *Orders*:

Restricting Field Values with Radio Controls (see page 1748)	TDBRadioGroup is a data-aware version of a radio group control. It enables you to set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group includes one button for each value a field can accept. Users can set the value for a data field by selecting the desired radio button. The *Items property determines the radio buttons that appear in the group. Items is a string list. One radio button is displayed for each string in Items, and each string appears to the... more (see page 1748)*
Disabling and Enabling Data Display (see page 1768)	When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker. *DisableControls is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's EnableControls method to re-enable display for the controls.* *Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a... more (see page 1768)*
Displaying and Editing Data in List and Combo Boxes (see page 1768)	There are four data controls that provide the user with a set of default data values to choose from at runtime. These are data-aware versions of standard list and combo box controls: • TDBListBox, which displays a scrollable list of items from which a user can choose to enter in a data field. A data-aware list box displays a default value for a field in the current record and highlights its corresponding entry in the list. If the current row's field value is not in the list, no value is highlighted in the list box. When a user selects a list... more (see page 1768)
Displaying and Editing Data in Lookup List and Combo Boxes (see page 1769)	Lookup list boxes and lookup combo boxes (TDBLookupListBox and TDBLookupComboBox) present the user with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset. For example, consider an order form whose fields are tied to the *OrdersTable*. *OrdersTable* contains a *CustNo* field corresponding to a customer ID, but *OrdersTable* does not have any other customer information. The *CustomersTable*, on the other hand, contains a *CustNo* field corresponding to a customer ID, and also contains additional information, such as... more (see page 1769)
Displaying and Editing Fields in an Edit Box (see page 1770)	TDBEdit is a data-aware version of an edit box component. *TDBEdit* displays the current value of a data field to which it is linked and permits it to be edited using standard edit box techniques. For example, suppose *CustomersSource* is a *TDataSource* component that is active and linked to an open TClientDataSet called *CustomersTable*. You can then place a *TDBEdit* component on a form and set its properties as follows: • *DataSource*: CustomersSource • *DataField*: CustNo The data-aware edit box component immediately displays the value of the current row of the *CustNo* column of the *CustomersTable* dataset, both at... more (see page 1770)
Displaying and Editing Graphics Fields in an Image Control (see page 1771)	TDBImage is a data-aware control that displays graphics contained in BLOB fields. By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the Clipboard using the CutToClipboard, CopyToClipboard, and PasteFromClipboard methods. You can, instead, supply your own editing methods attached to the event handlers for the control. By default, an image control displays as much of a graphic as fits in the control, cropping the image if it is too big. You can set the *Stretch property to True to resize the graphic to fit within an image control as it is... more (see page 1771)*

Displaying and Editing Text in a Memo Control (see page 1771)	TDBMemo is a data-aware component—similar to the standard TMemo component—that can display lengthy text data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display large text fields or text data contained in binary large object (BLOB) fields. By default, *TDBMemo* permits a user to edit memo text. To prevent editing, set the *ReadOnly property of the memo control to True. To display tabs and permit users to enter them in a memo, set the WantTabs property to True. To limit the number of characters users can... more (see page 1771)*
Displaying and Editing Text in a Rich Edit Memo Control (see page 1772)	TDBRichEdit is a data-aware component—similar to the standard *TRichEdit* component—that can display formatted text stored in a binary large object (BLOB) field. *TDBRichEdit* displays formatted, multi-line text, and permits a user to enter formatted multi-line text as well. **Note:** While *TDBRichEdit* provides properties and methods to enter and work with rich text, it does not provide any user interface components to make these formatting options available to the user. Your application must implement the user interface to surface rich text capabilities. By default, *TDBRichEdit* permits a user to edit memo text. To prevent editing, set the *ReadOnly property of the... more (see page 1772)*
Displaying Data as Labels (see page 1772)	TDBText is a read-only control similar to the TLabel component on the Standard category of the **Tool palette**. A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zip code field from a separate table. A *TDBText* component tied to the zip code table could... more (see page 1772)
Editing and Updating Data (see page 1773)	All data controls except the navigator display data from a database field. In addition, you can use them to edit and update data as long as the underlying dataset allows it. **Note:** Unidirectional datasets never permit users to edit and update data. The following topics describe how to allow users to edit data using data controls: • Enabling Editing in Controls On User Entry (see page 1774) • Editing Data in a Control (see page 1773)
Editing Data in a Control (see page 1773)	A data control can only post edits to its associated dataset if the dataset's CanModify property is *True*. *CanModify* is always *False* for unidirectional datasets. Some datasets have a *ReadOnly* property that lets you specify whether *CanModify* is *True*. **Note:** Whether a dataset can update data depends on whether the underlying database table permits updates. Even if the dataset's *CanModify* property is *True*, the Enabled property of the data source that connects the dataset to the control must be *True* as well before the control can post updates back to the database table. The *Enabled* property of... more (see page 1773)
Enabling and Disabling the Data Source (see page 1774)	The data source has an Enabled property that determines if it is connected to its dataset. When *Enabled* is *True*, the data source is connected to a dataset. You can temporarily disconnect a single data source from its dataset by setting *Enabled* to *False*. When *Enabled* is *False*, all data controls attached to the data source component go blank and become inactive until *Enabled* is set to *True*. It is recommended, however, to control access to a dataset through a dataset component's DisableControls and EnableControls methods because they affect all attached data sources.
Enabling Editing in Controls On User Entry (see page 1774)	A dataset must be in *dsEdit* state to permit editing to its data. If the data source's AutoEdit property is *True* (the default), the data control handles the task of putting the dataset into *dsEdit* mode as soon as the user tries to edit its data. If *AutoEdit* is *False*, you must provide an alternate mechanism for putting the dataset into edit mode. One such mechanism is to use a TDBNavigator control with an *Edit* button, which lets users explicitly put the dataset into edit mode. For more information about *TDBNavigator*, see Navigating and manipulating records (see page 1767). Alternately,... more (see page 1774)
Enabling Mouse, Keyboard, and Timer Events (see page 1774)	The Enabled property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is *True*. To prevent mouse, keyboard, or timer events from reaching a data control, set its *Enabled* property to *False*. When *Enabled* is *False*, the data source that connects the control to its dataset does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

Handling Boolean Field Values with Check Boxes (see page 1775)	TDBCheckBox is a data-aware check box control. It can be used to set the values of Boolean fields in a dataset. For example, a customer invoice form might have a check box control that when checked indicates the customer is tax-exempt, and when unchecked indicates that the customer is not tax-exempt. The data-aware check box control manages its checked or unchecked state by comparing the value of the current field to the contents of ValueChecked and ValueUnchecked properties. If the field value matches the *ValueChecked* property, the control is checked. Otherwise, if the field matches the *ValueUnchecked* property, the control... more (see page 1775)
Refreshing Data Display (see page 1776)	The Refresh method for a dataset flushes local buffers and re-fetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application. If you are using cached updates, before you refresh the dataset you must apply any updates the dataset has currently cached. Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls... more (see page 1776)
Responding to Changes Mediated by the Data Source (see page 1776)	Because the data source provides the link between the data control and its dataset, it mediates all of the communication that occurs between the two. Typically, the data-aware control automatically responds to changes in the dataset. However, if your user interface is using controls that are not data-aware, you can use the events of a data source component to manually provide the same sort of response. The OnDataChange event occurs whenever the data in a record may have changed, including field edits or when the cursor moves to a new record. This event is useful for making sure the control... more (see page 1776)
Using Common Data Control Features (see page 1777)	The following tasks are common to most data controls: • Associating a data control with a dataset (see page 1746) • Editing and updating data (see page 1773) • Disabling and enabling data display (see page 1768) • Refreshing data display (see page 1776) • Enabling mouse (see page 1774) Data controls let you display and edit fields of data associated with the current record in a dataset. The following table summarizes the data controls that appear on the Data Controls category of the **Tool palette**. ***Data controls***
Using TDBListBox and TDBComboBox (see page 1778)	When using *TDBListBox* or *TDBComboBox*, you must use the String List editor at design time to create the list of items to display. To bring up the String List editor, click the ellipsis button for the *Items property in the **Object Inspector**. Then type in the items that you want to have appear in the list. At runtime, use the methods of the Items property to manipulate its string list.* *When a TDBListBox or TDBComboBox control is linked to a field through its DataField property, the field value appears selected in the list. If the current value is... more (see page 1778)*
Using Data Controls (see page 1778)	The Data Controls category of the **Tool palette** provides a set of data-aware controls that represent data from fields in a database record, and, if the dataset allows it, enable users to edit that data and post changes back to the database. By placing data controls onto the forms in your database application, you can build your database application's user interface (UI) so that information is visible and accessible to users. The data-aware controls you add to your user interface depend on several factors, including the following: • The type of data you are displaying. You can choose between controls... more (see page 1778)

3.2.3.9.1 Associating a Data Control with a Dataset

Data controls connect to datasets by using a data source. A data source component (TDataSource) acts as a conduit between

the control and a dataset containing data. Each data-aware control must be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form.

Note: Data source components are also required for linking unnested datasets in master-detail relationships.

To associate a data control with a dataset

1. Place a dataset in a data module (or on a form), and set its properties as appropriate.
2. Place a data source in the same data module (or form). Using the **Object Inspector**, set its DataSet property to the dataset you placed in step 1.
3. Place a data control from the Data Access category of the **Tool palette** onto a form.
4. Using the **Object Inspector**, set the *DataSource* property of the control to the data source component you placed in step 2.
5. Set the *DataField* property of the control to the name of a field to display, or select a field name from the drop-down list for the property. This step does not apply to *TDBGrid, TDBCtrlGrid,* and *TDBNavigator* because they access all available fields in the dataset.
6. Set the *Active* property of the dataset to *True* to display data in the control.

For more information about managing the link between the data control and its dataset, see

- Changing the Associated Dataset at Runtime (see page 1747)
- Enabling and Disabling the Data Source (see page 1774)
- Responding to Changes Mediated by the Data Source (see page 1776)

See Also

Editing and Updating Data (see page 1773)

Disabling and Enabling Data Display (see page 1768)

Refreshing Data Display (see page 1776)

Enabling Mouse (see page 1774)

3.2.3.9.2 Choosing How to Organize the Data

When you build the user interface for your database application, you have choices to make about how you want to organize the display of information and the controls that manipulate that information.

One of the first decisions to make is whether you want to display a single record at a time, or multiple records (see page 1760).

In addition, you will want to add controls to navigate and manipulate records (see page 1767). The TDBNavigator control provides built-in support for many of the functions you may want to perform.

See Also

Using Common Data Control Features (see page 1777)

Navigating and Manipulating Records (see page 1767)

3.2.3.9.3 Changing the Associated Dataset at Runtime

In Associating a Data Control with a Dataset (see page 1746), the datasource was associated with its dataset by setting the *DataSet* property at design time. At runtime, you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the *CustSource* data source component between the dataset components named *Customers* and *Orders*:

```
with CustSource do begin
  if (DataSet = Customers) then
    DataSet := Orders
  else
    DataSet := Customers;
end;
if (CustSource->DataSet == Customers)
  CustSource->DataSet = Orders;
else
  CustSource->DataSet = Customers;
```

You can also set the *DataSet* property to a dataset on another form to synchronize the data controls on two forms. For example:

```
procedure TForm2.FormCreate (Sender : TObject);
begin
  DataSource1.Dataset := Form1.Table1;
end;
void __fastcall TForm2::FormCreate(TObject *Sender)
{
  DataSource1->DataSet = Form1->Table1;
}
```

See Also

Enabling and Disabling the Data Source (see page 1774)

Responding to Changes Mediated by the Data Source (see page 1776)

3.2.3.9.4 Restricting Field Values with Radio Controls

TDBRadioGroup is a data-aware version of a radio group control. It enables you to set the value of a data field with a radio button control where there is a limited number of possible values for the field. The radio group includes one button for each value a field can accept. Users can set the value for a data field by selecting the desired radio button.

The *Items property determines the radio buttons that appear in the group. Items is a string list. One radio button is displayed for each string in Items, and each string appears to the right of a radio button as the button's label.*

If the current value of a field associated with a radio group matches one of the strings in the Items property, that radio button is selected. For example, if three strings, "Red," "Yellow," and "Blue," are listed for Items, and the field for the current record contains the value "Blue," then the third button in the group appears selected.

***Note:** If the field does not match any strings in Items, a radio button may still be selected if the field matches a string in the Values property. If the field for the current record does not match any strings in Items or Values, no radio button is selected.*

The Values property can contain an optional list of strings that can be returned to the dataset when a user selects a radio button and posts a record. Strings are associated with buttons in numeric sequence. The first string is associated with the first button, the second string with the second button, and so on. For example, suppose Items contains "Red," "Yellow," and "Blue," and Values contains "Magenta," "Yellow," and "Cyan." If a user selects the button labeled "Red," "Magenta" is posted to the database.

If strings for Values are not provided, the Item string for a selected radio button is returned to the database when a record is posted.

Topics

Name	Description
Arranging the Order of Persistent Columns (see page 1751)	The order in which columns appear in the Columns editor is the same as the order the columns appear in the grid. You can change the column order by dragging and dropping columns within the Columns list box.
Choosing Navigator Buttons to Display (see page 1752)	When you first place a TDBNavigator on a form at design time, all its buttons are visible. You can use the VisibleButtons property to turn off buttons you do not want to use on a form. For example, when working with a unidirectional dataset, only the *First*, *Next*, and *Refresh* buttons are meaningful. On a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

Controlling Grid Drawing (see page 1753)	Your first level of control over how a grid control draws itself is setting column properties. The grid automatically uses the font, color, and alignment properties of a column to draw the cells of that column. The text of data fields is drawn using the *DisplayFormat* or *EditFormat properties of the field component associated with the column.* *You can augment the default grid display logic with code in a grid's OnDrawColumnCell event. If the grid's DefaultDrawing property is True, all the normal drawing is performed before your OnDrawColumnCell event handler is called. Your code can then draw on top of... more (see page 1753)*
Creating a Customized Grid (see page 1754)	A customized grid is one for which you define persistent column objects that describe how a column appears and how the data in the column is displayed. A customized grid lets you configure multiple grids to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example). A customized grid also enables you to let users modify the appearance of the grid at runtime without affecting the fields used by the grid or the field order of the dataset. Customized grids are best used with datasets whose structure is known... more (see page 1754)
Creating a Grid That Contains Other Data-aware Controls (see page 1755)	A TDBCtrlGrid control displays multiple fields in multiple records in a tabular grid format. Each cell in a grid displays multiple fields from a single row.
Creating Persistent Columns (see page 1756)	To customize the appearance of grid at design time, you invoke the Columns editor to create a set of persistent column objects for the grid. At runtime, the State property for a grid with persistent column objects is automatically set to *csCustomized*.
Defining a Lookup List Column (see page 1757)	You can create a column that displays a drop-down list of values, similar to a lookup combo box control (see page 1769). To specify that the column acts like a combo box, set the column's *ButtonStyle property to cbsAuto. Once you populate the list with values, the grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode.* *There are two ways to populate that list with the values for users to select:* • You can fetch the values from a lookup table. To make a column display a drop-down list of values drawn from... more (see page 1757)
Deleting Persistent Columns (see page 1757)	Deleting a persistent column from a grid is useful for eliminating fields that you do not want to display.
Displaying ADT and Array Fields (see page 1758)	Sometimes the fields of the grid's dataset do not represent simple values such as text, graphics, numerical values, and so on. Some database servers allow fields that are a composite of simpler data types, such as ADT fields or array fields. There are two ways a grid can display composite fields: • It can "flatten out" the field so that each of the simpler types that make up the field appears as a separate field in the dataset. • It can display composite fields in a single column, reflecting the fact that they are a single field. When a composite field is... more (see page 1758)
Displaying Fly-over Help (see page 1759)	To display fly-over help for each navigator button at runtime, set the navigator ShowHint property to True. When *ShowHint* is True, the navigator displays fly-by Help hints whenever you pass the mouse cursor over the navigator buttons. *ShowHint* is *False* by default. The Hints property controls the fly-over help text for each button. By default *Hints* is an empty string list. When *Hints* is empty, each navigator button displays default help text. To provide customized fly-over help for the navigator buttons, use the String list editor to enter a separate line of hint text for each button in the *Hints... more (see page 1759)*
Displaying Multiple Records (see page 1760)	Sometimes you want to display many records in the same form. For example, an invoicing application might show all the orders made by a single customer on the same form. To display multiple records, use a grid control. Grid controls provide a multi-field, multi-record view of data that can make your application's user interface more compelling and effective. They are discussed in Viewing and editing data with TDBGrid (see page 1766) and Creating a grid that contains other data-aware controls (see page 1755). **Note:** You can't display multiple records when using a unidirectional dataset. You may want to design a user interface that displays both... more (see page 1760)

Editing in the Grid (see page 1760)	At runtime, you can use a grid to modify existing data and enter new records, if the following default conditions are met: • The CanModify property of the *Dataset* is *True*. • The ReadOnly property of grid is *False*. When a user edits a record in the grid, changes to each field are posted to an internal record buffer, but are not posted until the user moves to a different record in the grid. Even if focus is changed to another control on a form, the grid does not post changes until another the cursor for the dataset is moved... more (see page 1760)
Putting a Button in a Column (see page 1761)	A column can display an ellipsis button (...) to the right of the normal cell editor. `Ctrl+Enter` or a mouse click fires the grid's OnEditButtonClick event. You can use the ellipsis button to bring up forms containing more detailed views of the data in the column. For example, in a table that displays summaries of invoices, you could set up an ellipsis button in the invoice total column to bring up a form that displays the items in that invoice, or the tax calculation method, and so on. For graphic fields, you could use the ellipsis button to bring up... more (see page 1761)
Responding to User Actions at Runtime (see page 1761)	You can modify grid behavior by writing event handlers to respond to specific actions within the grid at runtime. Because a grid typically displays many fields and records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column. The following table lists the grid events available in the **Object Inspector**. ***Grid control events***
Restoring Default Values to a Column (see page 1762)	At runtime you can test a column's AssignedValues property to determine whether a column property has been explicitly assigned. Values that are not explicitly defined are dynamically based on the associated field or the grid's defaults. You can undo property changes made to one or more columns. In the Columns editor, select the column or columns to restore, and then select Restore Defaults from the context menu. Restore defaults discards assigned property settings and restores a column's properties to those derived from its underlying field component At runtime, you can reset all default properties for a single column by calling... more (see page 1762)
Setting Column Properties at Design Time (see page 1763)	Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component (called the *default source*) such as a grid or an associated field component. To set a column's properties, select the column in The Columns editor and set its properties in the **Object Inspector**. The following table summarizes key column properties you can set. ***Column properties***
Setting Grid Options (see page 1764)	You can use the grid Options property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the *Options* property in the **Object Inspector** is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of Boolean properties that you can set individually. To view and set these properties, click on the + sign. The list of options in the **Object Inspector** below the *Options* property. The + sign changes to a –(minus) sign, that collapses... more (see page 1764)
Using a Grid Control in Its Default State (see page 1765)	The *Stateproperty of the grid's Columns property indicates whether persistent column objects exist for the grid. Columns.State is a runtime-only property that is automatically set for a grid. The default state is csDefault, meaning that persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined primarily by the properties of the fields in the grid's dataset, or, if there are no persistent field components, by a default set of display characteristics.* *When the grid's Columns.State property is csDefault, grid columns are dynamically generated from the visible... more (see page 1765)*

Using a Single Navigator for Multiple Datasets (see page 1766)	As with other data-aware controls, a navigator's *DataSource property specifies the data source that links the control to a dataset. By changing a navigator's DataSource property at runtime, a single navigator can provide record navigation and manipulation for multiple datasets.* *Suppose a form contains two edit controls linked to the CustomersTable and OrdersTable datasets through the CustomersSource and OrdersSource data sources respectively. When a user enters the edit control connected to CustomersSource, the navigator should also use CustomersSource, and when the user enters the edit control connected to OrdersSource, the navigator should switch to OrdersSource as well.... more (see page 1766)*
Viewing and Editing Data with TDBGrid (see page 1766)	A TDBGrid control lets you view and edit records in a dataset in a tabular grid format. Three factors affect the appearance of records displayed in a grid control: • Existence of persistent column objects defined for the grid using the Columns editor. Persistent column objects provide great flexibility setting grid and data appearance. For information on using persistent columns, see Creating a customized grid (see page 1754). • Creation of persistent field components for the dataset displayed in the grid. For more information about creating persistent field components using the Fields editor, see Working with field components (see page 1877). • The dataset's *ObjectView* property setting... more (see page 1766)
Navigating and Manipulating Records (see page 1767)	TDBNavigator provides users a simple control for navigating through records in a dataset, and for manipulating records. The navigator consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display. The following figure shows the navigator that appears by default when you place it on a form at design time. The navigator consists of a series of buttons that... more (see page 1767)

See Also

Displaying Data as Labels (see page 1772)

Displaying and Editing Fields in an Edit Box (see page 1770)

Displaying and Editing Text in a Memo Control (see page 1771)

Displaying and Editing Text in a Rich Edit Memo Control (see page 1772)

Displaying and Editing Graphics Fields in an Image Control (see page 1771)

Displaying and Editing Data in List and Combo Boxes (see page 1768)

Handling Boolean Field Values with Check Boxes (see page 1775)

3.2.3.9.4.1 Arranging the Order of Persistent Columns

The order in which columns appear in the Columns editor is the same as the order the columns appear in the grid. You can change the column order by dragging and dropping columns within the Columns list box.

To change the order of a column

1. Select the column in the Columns list box.
2. Drag it to a new location in the list box.

You can also change the column order at runtime by clicking on the column title and dragging the column to a new position.

Note: Reordering persistent fields in the Fields editor also reorders columns in a default grid, but not a custom grid.

Warning: You cannot reorder columns in grids containing both dynamic columns and dynamic fields at design time, since there is nothing persistent to record the altered field or column order.

At runtime, a user can use the mouse to drag a column to a new location in the grid if its DragMode property is set to *dmManual*. Reordering the columns of a grid with a *State* property of *csDefault* state also reorders field components in the dataset underlying the grid. The order of fields in the physical table is not affected. To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

At runtime, the grid's *OnColumnMoved event fires after a column has been moved.*

See Also

Creating Persistent Columns (see page 1756)

Deleting Persistent Columns (see page 1757)

Defining a Lookup List Column (see page 1757)

Putting a Button in a Column (see page 1761)

Setting Column Properties at Design Time (see page 1763)

Restoring Default Values to a Column (see page 1762)

3.2.3.9.4.2 Choosing Navigator Buttons to Display

When you first place a TDBNavigator on a form at design time, all its buttons are visible. You can use the VisibleButtons property to turn off buttons you do not want to use on a form. For example, when working with a unidirectional dataset, only the *First*, *Next*, and *Refresh* buttons are meaningful. On a form that is intended for browsing rather than editing, you might want to disable the *Edit*, *Insert*, *Delete*, *Post*, and *Cancel* buttons.

Hiding and showing navigator buttons at design time

The VisibleButtons property in the **Object Inspector** is displayed with a + sign to indicate that it can be expanded to display a Boolean value for each button on the navigator. To view and set these values, click on the + sign. The list of buttons that can be turned on or off appears in the **Object Inspector** below the *VisibleButtons* property. The + sign changes to a –(minus) sign, which you can click to collapse the list of properties.

Button visibility is indicated by the *Boolean* state of the button value. If a value is set to *True*, the button appears in the TDBNavigator. If *False*, the button is removed from the navigator at design time and runtime.

Note: As button values are set to *False*, they are removed from the TDBNavigator on the form, and the remaining buttons are expanded in width to fill the control. You can drag the control's handles to resize the buttons.

Hiding and showing navigator buttons at runtime

At runtime you can hide or show navigator buttons in response to user actions or application states. For example, suppose you provide a single navigator for navigating through two different datasets, one of which permits users to edit records, and the other of which is read-only. When you switch between datasets, you want to hide the navigator's *Insert*, *Delete*, *Edit*, *Post*, *Cancel*, and *Refresh* buttons for the read-only dataset, and show them for the other dataset.

For example, suppose you want to prevent edits to the *OrdersTable* by hiding the *Insert*, *Delete*, *Edit*, *Post*, *Cancel*, and *Refresh* buttons on the navigator, but that you also want to allow editing for the *CustomersTable*. The VisibleButtons property controls which buttons are displayed in the navigator. Here's one way you might code the event handler:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
  begin
    DBNavigatorAll.DataSource := CustomerCompany.DataSource;
    DBNavigatorAll.VisibleButtons := [nbFirst,nbPrior,nbNext,nbLast];
  end
```

```
  else
  begin
    DBNavigatorAll.DataSource := OrderNum.DataSource;
    DBNavigatorAll.VisibleButtons := DBNavigatorAll.VisibleButtons + [nbInsert,
      nbDelete,nbEdit,nbPost,nbCancel,nbRefresh];
  end;
end;
```

For a C++ example, suppose you want to prevent edits to the OrdersTable by hiding the Insert, Delete, Edit, Post, Cancel, and Refresh buttons on the navigator, but that you also want to allow editing for the CustomersTable. The VisibleButtons property controls which buttons are displayed in the navigator. Here's one way you might code the event handler:

```
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
  if (Sender == (TObject *)CustomerCompany)
  {
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
    DBNavigatorAll->VisibleButtons = DBNavigatorAll->VisibleButtons
<< nbFirst << nbPrior << nbNext << nbLast;
  }
  else
  {
    DBNavigatorAll->DataSource = OrderNum->DataSource;
    DBNavigatorAll->VisibleButtons = DBNavigatorAll->VisibleButtons << nbInsert
<< nbDelete << nbEdit
<< nbPost << nbCancel << nbRefresh;
  }
}
```

See Also

Displaying Fly-over Help (see page 1759)

Using a Single Navigator for Multiple Datasets (see page 1766)

3.2.3.9.4.3 Controlling Grid Drawing

Your first level of control over how a grid control draws itself is setting column properties. The grid automatically uses the font, color, and alignment properties of a column to draw the cells of that column. The text of data fields is drawn using the *DisplayFormat* or *EditFormat properties of the field component associated with the column.*

You can augment the default grid display logic with code in a grid's OnDrawColumnCell event. If the grid's DefaultDrawing property is True, all the normal drawing is performed before your OnDrawColumnCell event handler is called. Your code can then draw on top of the default display. This is primarily useful when you have defined a blank persistent column and want to draw special graphics in that column's cells.

If you want to replace the drawing logic of the grid entirely, set DefaultDrawing to False and place your drawing code in the grid's OnDrawColumnCell event. If you want to replace the drawing logic only in certain columns or for certain field data types, you can call the DefaultDrawColumnCell inside your OnDrawColumnCell event handler to have the grid use its normal drawing code for selected columns. This reduces the amount of work you have to do if you only want to change the way Boolean field types are drawn, for example.

See Also

Using a Grid Control in Its Default State (see page 1765)

Creating a Customized Grid (see page 1754)

Creating Persistent Columns (see page 1756)

Setting Grid Options (see page 1764)

Editing in the Grid (see page 1760)

Responding to User Actions at Runtime (see page 1761)

Displaying ADT and Array Fields (see page 1758)

3.2.3.9.4.4 Creating a Customized Grid

A customized grid is one for which you define persistent column objects that describe how a column appears and how the data in the column is displayed. A customized grid lets you configure multiple grids to present different views of the same dataset (different column orders, different field choices, and different column colors and fonts, for example). A customized grid also enables you to let users modify the appearance of the grid at runtime without affecting the fields used by the grid or the field order of the dataset.

Customized grids are best used with datasets whose structure is known at design time. Because they expect field names established at design time to exist in the dataset, customized grids are not well suited to browsing arbitrary tables selected at runtime.

Understanding persistent columns

When you create persistent column objects for a grid, they are only loosely associated with underlying fields in a grid's dataset. Default property values for persistent columns are dynamically fetched from a default source (the associated field or the grid itself) until a value is assigned to the column property. Until you assign a column property a value, its value changes as its default source changes. Once you assign a value to a column property, it no longer changes when its default source changes.

For example, the default source for a column title caption is an associated field's DisplayLabel property. If you modify the *DisplayLabelproperty, the column title reflects that change immediately. If you then assign a string to the column title's caption, the tile caption becomes independent of the associated field's DisplayLabel property. Subsequent changes to the field's DisplayLabel property no longer affect the column's title.*

Persistent columns exist independently from field components with which they are associated. In fact, persistent columns do not have to be associated with field objects at all. If a persistent column's FieldName property is blank, or if the field name does not match the name of any field in the grid's current dataset, the column's Field property is NULL and the column is drawn with blank cells. If you override the cell's default drawing method, you can display your own custom information in the blank cells. For example, you can use a blank column to display aggregated values on the last record of a group of records that the aggregate summarizes. Another possibility is to display a bitmap or bar chart that graphically depicts some aspect of the record's data.

Two or more persistent columns can be associated with the same field in a dataset. For example, you might display a part number field at the left and right extremes of a wide grid to make it easier to find the part number without having to scroll the grid.

***Note:** Because persistent columns do not have to be associated with a field in a dataset, and because multiple columns can reference the same field, a customized grid's FieldCount property can be less than or equal to the grid's column count. Also note that if the currently selected column in a customized grid is not associated with a field, the grid's SelectedField property is NULL and the SelectedIndexproperty is –1.*

Persistent columns can be configured to display grid cells as a combo box drop-down list of lookup values from another dataset or from a static pick list, or as an ellipsis button (...) in a cell that can be clicked upon to launch special data viewers or dialogs related to the current cell.

The following topics provide additional information about persistent columns:

- Creating Persistent Columns (see page 1756)
- Deleting Persistent Columns (see page 1757)
- Arranging the Order of Persistent Columns (see page 1751)
- Setting Column Properties at Design Time (see page 1763)
- Defining a Lookup List Column (see page 1757)
- Putting a Button in a Column (see page 1761)

- Restoring Default Values to a Column (see page 1762)

See Also

Using a Grid Control in Its Default State (see page 1765)

Setting Grid Options (see page 1764)

Editing in the Grid (see page 1760)

Displaying ADT and Array Fields (see page 1758)

Controlling Grid Drawing (see page 1753)

Responding to User Actions at Runtime (see page 1761)

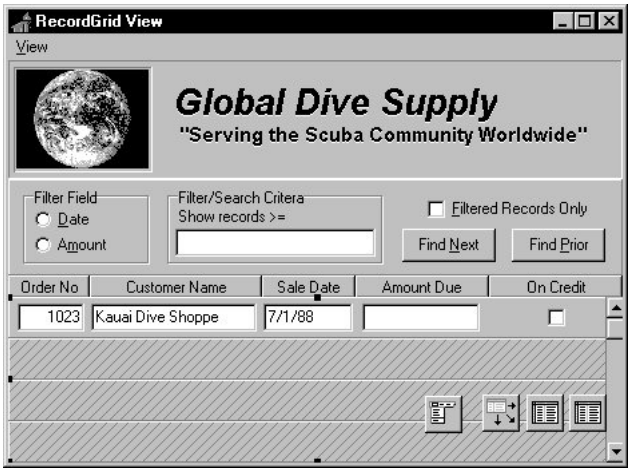
3.2.3.9.4.5 Creating a Grid That Contains Other Data-aware Controls

A TDBCtrlGrid control displays multiple fields in multiple records in a tabular grid format. Each cell in a grid displays multiple fields from a single row.

To use a database control grid

1. Place a database control grid on a form.
2. Set the grid's DataSource property to the name of a data source.
3. Place individual data controls within the design cell for the grid. The design cell for the grid is the top or leftmost cell in the grid, and is the only cell into which you can place other controls.
4. Set the *DataField* property for each data control to the name of a field. The data source for these data controls is already set to the data source of the database control grid.
5. Arrange the controls within the cell as desired.

When you compile and run an application containing a database control grid, the arrangement of data controls you set in the design cell at runtime is replicated in each cell of the grid. Each cell displays a different record in a dataset.



The following table summarizes some of the unique properties for database control grids that you can set at design time:

Selected database control grid properties

Property	Purpose
AllowDelete	*True* (default): Permits record deletion. *False*: Prevents record deletion.

AllowInsert	*True* (default): Permits record insertion. *False*: Prevents record insertion.
ColCount	Sets the number of columns in the grid. Default = 1.
Orientation	*goVertical* (default): Display records from top to bottom. *goHorizontal*: Displays records from left to right.
PanelHeight	Sets the height for an individual panel. Default = 72.
PanelWidth	Sets the width for an individual panel. Default = 200.
RowCount	Sets the number of panels to display. Default = 3.
ShowFocus	*True* (default): Displays a focus rectangle around the current record's panel at runtime. *False*: Does not display a focus rectangle.

See Also

Viewing and Editing Data with TDBGrid (see page 1766)

3.2.3.9.4.6 Creating Persistent Columns

To customize the appearance of grid at design time, you invoke the Columns editor to create a set of persistent column objects for the grid. At runtime, the State property for a grid with persistent column objects is automatically set to *csCustomized*.

To create persistent columns for a grid control

1. Select the grid component in the form.
2. Invoke the Columns editor by double clicking on the grid's *Columns property in the* ***Object Inspector****.*

The Columns list box displays the persistent columns that have been defined for the selected grid. When you first bring up the Columns editor, this list is empty because the grid is in its default state, containing only dynamic columns.

You can create persistent columns for all fields in a dataset at once, or you can create persistent columns on an individual basis.

To create persistent columns for all fields

1. Right-click the grid to invoke the context menu and choose Add All Fields. Note that if the grid is not already associated with a data source, Add All Fields is disabled. Associate the grid with a data source that has an active dataset before choosing Add All Fields.
2. If the grid already contains persistent columns, a dialog box asks if you want to delete the existing columns, or append to the column set. If you choose Yes, any existing persistent column information is removed, and all fields in the current dataset are inserted by field name according to their order in the dataset. If you choose No, any existing persistent column information is retained, and new column information, based on any additional fields in the dataset, are appended to the dataset.
3. Click Close to apply the persistent columns to the grid and close the dialog box.

To create persistent columns individually

1. Choose the Add button in the Columns editor. The new column will be selected in the list box. The new column is given a sequential number and default name (for example, 0 - TColumn).
2. To associate a field with this new column, set the FieldName property in the **Object Inspector**.
3. To set the title for the new column, expand the Title property in the **Object Inspector** and set its *Caption* property.
4. Close the Columns editor to apply the persistent columns to the grid and close the dialog box.

At runtime, you can switch to persistent columns by assigning *csCustomized* to the *Columns.State* property. Any existing columns in the grid are destroyed and new persistent columns are built for each field in the grid's dataset. You can then add a persistent column at runtime by calling the *Add* method for the column list:

```
DBGrid1.Columns.Add;
```

```
DBGrid1->Columns->Add();
```

See Also

Deleting Persistent Columns (see page 1757)

Arranging the Order of Persistent Columns (see page 1751)

Defining a Lookup List Column (see page 1757)

Putting a Button in a Column (see page 1761)

Setting Column Properties at Design Time (see page 1763)

Restoring Default Values to a Column (see page 1762)

3.2.3.9.4.7 Defining a Lookup List Column

You can create a column that displays a drop-down list of values, similar to a lookup combo box control (see page 1769). To specify that the column acts like a combo box, set the column's *ButtonStyle property to cbsAuto. Once you populate the list with values, the grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode.*

There are two ways to populate that list with the values for users to select:

- You can fetch the values from a lookup table. To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field (see page 1868) in the dataset. Once the lookup field is defined, set the column's *FieldName to the lookup field name. The drop-down list is automatically populated with lookup values defined by the lookup field.*
- You can specify a list of values explicitly at design time. To enter the list values at design time, double-click the *PickList property for the column in the **Object Inspector**. This brings up the String List editor, where you can enter the values that populate the pick list for the column.*

By default, the drop-down list displays 7 values. You can change the length of this list by setting the DropDownRows property.

Note: To restore a column with an explicit pick list to its normal behavior, delete all the text from the pick list using the String List editor.

See Also

Creating Persistent Columns (see page 1756)

Deleting Persistent Columns (see page 1757)

Arranging the Order of Persistent Columns (see page 1751)

Putting a Button in a Column (see page 1761)

Setting Column Properties at Design Time (see page 1763)

Restoring Default Values to a Column (see page 1762)

3.2.3.9.4.8 Deleting Persistent Columns

Deleting a persistent column from a grid is useful for eliminating fields that you do not want to display.

To remove a persistent column from a grid

1. Double-click the grid to display the Columns editor.
2. Select the field to remove in the Columns list box.
3. Click Delete (you can also use the context menu or `Del` key, to remove a column).

   Note: If you delete all the columns from a grid, the *Columns.State* property reverts to its *csDefault* state and automatically build dynamic columns for each field in the dataset.

You can delete a persistent column at runtime by simply freeing the column object:

```
DBGrid1.Columns[5].Free;
delete DBGrid1->Columns->Items[5];
```

See Also

Creating Persistent Columns (see page 1756)

Arranging the Order of Persistent Columns (see page 1751)

Defining a Lookup List Column (see page 1757)

Putting a Button in a Column (see page 1761)

Setting Column Properties at Design Time (see page 1763)

Restoring Default Values to a Column (see page 1762)

3.2.3.9.4.9 Displaying ADT and Array Fields

Sometimes the fields of the grid's dataset do not represent simple values such as text, graphics, numerical values, and so on. Some database servers allow fields that are a composite of simpler data types, such as ADT fields or array fields.

There are two ways a grid can display composite fields:

- It can "flatten out" the field so that each of the simpler types that make up the field appears as a separate field in the dataset.
- It can display composite fields in a single column, reflecting the fact that they are a single field.

When a composite field is flattened out, its constituents appear as separate fields that reflect their common source only in that each field name is preceded by the name of the common parent field in the underlying database table.

To display composite fields as if they were flattened out, set the dataset's ObjectView property to *False*. The dataset stores composite fields as a set of separate fields, and the grid reflects this by assigning each constituent part a separate column.

When displaying composite fields in a single column, the column can be expanded and collapsed by clicking on the arrow in the title bar of the field, or by setting the *Expa*nded property of the column:

- When a column is expanded, each child field appears in its own sub-column with a title bar that appears below the title bar of the parent field. That is, the title bar for the grid increases in height, with the first row giving the name of the composite field, and the second row subdividing that for the individual parts. Fields that are not composites appear with title bars that are extra high. This expansion continues for constituents that are in turn composite fields (for example, a detail table nested in a detail table), with the title bar growing in height accordingly.
- When the field is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

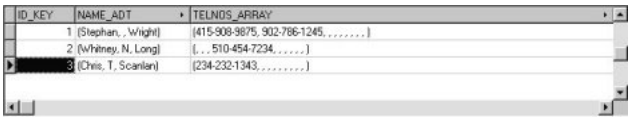
To display a composite field in an expanding and collapsing column, set the dataset's *ObjectView* property to *True*. The dataset stores the composite field as a single field component that contains a set of nested sub-fields. The grid reflects this in a column that can expand or collapse

The following figure shows a grid with an ADT field and an array field. The dataset's *ObjectView* property is set to *False* so that each child field has a column.



TDBGrid control with *ObjectView* set to *False*

The following figures show the grid with an ADT field and an array field. The first figure shows the fields collapsed. In this state they cannot be edited. The second figure shows the fields expanded. The fields are expanded and collapsed by clicking on the arrow in the fields title bar.



TDBGrid control with *Expanded* set to *False*



TDBGrid control with *Expanded* set to *True*

The following table lists the properties that affect the way ADT and array fields appear in a *TDBGrid*:

Properties that affect the way composite fields appear

Property	Object	Purpose
Expandable	TColumn	Indicates whether the column can be expanded to show child fields in separate, editable columns. (read-only)
Expanded	TColumn	Specifies whether the column is expanded.
MaxTitleRows	TDBGrid	Specifies the maximum number of title rows that can appear in the grid
ObjectView	TDataSet	Specifies whether fields are displayed flattened out, or in object mode, where each object field can be expanded and collapsed.
ParentColumn	TColumn	Refers to the TColumn object that owns the child field's column.

Note: In addition to ADT and array fields, some datasets include fields that refer to another dataset (dataset fields) or a record in another dataset (reference) fields. Data-aware grids display such fields as "(DataSet)" or "(Reference)", respectively. At runtime an ellipsis button appears to the right. Clicking on the ellipsis brings up a new form with a grid displaying the contents of the field. For dataset fields, this grid displays the dataset that is the field's value. For reference fields, this grid contains a single row that displays the record from another dataset.

See Also

Using Object Fields (see page 1862)

Creating a Customized Grid (see page 1754)

Using a Grid Control in Its Default State (see page 1765)

Setting Grid Options (see page 1764)

Editing in the Grid (see page 1760)

Controlling Grid Drawing (see page 1753)

Responding to User Actions at Runtime (see page 1761)

Working with Dataset Fields (see page 1867)

Working with reference fields (see page 1867)

3.2.3.9.4.10 Displaying Fly-over Help

To display fly-over help for each navigator button at runtime, set the navigator ShowHint property to True. When *ShowHint* is True, the navigator displays fly-by Help hints whenever you pass the mouse cursor over the navigator buttons. *ShowHint* is *False* by default.

The Hints property controls the fly-over help text for each button. By default *Hints* is an empty string list. When *Hints* is empty, each navigator button displays default help text. To provide customized fly-over help for the navigator buttons, use the String list

editor to enter a separate line of hint text for each button in the *Hints* property. When present, the strings you provide override the default hints provided by the navigator control.

See Also

Choosing Navigator Buttons to Display (see page 1752)

Using a Single Navigator for Multiple Datasets (see page 1766)

3.2.3.9.4.11 Displaying Multiple Records

Sometimes you want to display many records in the same form. For example, an invoicing application might show all the orders made by a single customer on the same form.

To display multiple records, use a grid control. Grid controls provide a multi-field, multi-record view of data that can make your application's user interface more compelling and effective. They are discussed in Viewing and editing data with TDBGrid (see page 1766) and Creating a grid that contains other data-aware controls (see page 1755).

Note: You can't display multiple records when using a unidirectional dataset.

You may want to design a user interface that displays both fields from a single record and grids that represent multiple records. There are two models that combine these two approaches:

- **Master-detail forms**: You can represent information from both a master table and a detail table by including both controls that display a single field and grid controls. For example, you could display information about a single customer with a detail grid that displays the orders for that customer. For information about linking the underlying tables in a master-detail form, see Creating Master/detail Relationships (see page 1590) and Establishing master/detail relationships using parameters (see page 1593).
- **Drill-down forms**: In a form that displays multiple records, you can include single field controls that display detailed information from the current record only. This approach is particularly useful when the records include long memos or graphic information. As the user scrolls through the records of the grid, the memo or graphic updates to represent the value of the current record. Setting this up is very easy. The synchronization between the two displays is automatic if the grid and the memo or image control share a common data source.

  Tip: It is generally not a good idea to combine these two approaches on a single form. It is usually confusing for users to understand the data relationships in such forms.

See Also

Displaying a Single Record

Analyzing Data (see page 1567)

Writing Reports (see page 1568)

3.2.3.9.4.12 Editing in the Grid

At runtime, you can use a grid to modify existing data and enter new records, if the following default conditions are met:

- The CanModify property of the *Dataset* is *True*.
- The ReadOnly property of grid is *False*.

When a user edits a record in the grid, changes to each field are posted to an internal record buffer, but are not posted until the user moves to a different record in the grid. Even if focus is changed to another control on a form, the grid does not post changes until another the cursor for the dataset is moved to another record. When a record is posted, the dataset checks all associated data-aware components for a change in status. If there is a problem updating any fields that contain modified data, the grid raises an exception, and does not modify the record.

Note: If your application caches updates, posting record changes only adds them to an internal cache. They are not posted back to the underlying database table until your application applies the updates.

You can cancel all edits for a record by pressing Esc in any field before moving to another record.

See Also

Using a Grid Control in Its Default State (see page 1765)

Creating a Customized Grid (see page 1754)

Setting Grid Options (see page 1764)

Displaying ADT and Array Fields (see page 1758)

Controlling Grid Drawing (see page 1753)

Responding to User Actions at Runtime (see page 1761)

3.2.3.9.4.13 Putting a Button in a Column

A column can display an ellipsis button (...) to the right of the normal cell editor. `Ctrl+Enter` or a mouse click fires the grid's OnEditButtonClick event. You can use the ellipsis button to bring up forms containing more detailed views of the data in the column. For example, in a table that displays summaries of invoices, you could set up an ellipsis button in the invoice total column to bring up a form that displays the items in that invoice, or the tax calculation method, and so on. For graphic fields, you could use the ellipsis button to bring up a form that displays an image.

To create an ellipsis button in a column

1. Select the column in the *Columns* list box.
2. Set *ButtonStyle* to *cbsEllipsis*.
3. Write an *OnEditButtonClick* event handler.

See Also

Creating Persistent Columns (see page 1756)

Deleting Persistent Columns (see page 1757)

Arranging the Order of Persistent Columns (see page 1751)

Defining a Lookup List Column (see page 1757)

Setting Column Properties at Design Time (see page 1763)

Restoring Default Values to a Column (see page 1762)

3.2.3.9.4.14 Responding to User Actions at Runtime

You can modify grid behavior by writing event handlers to respond to specific actions within the grid at runtime. Because a grid typically displays many fields and records at once, you may have very specific needs to respond to changes to individual columns. For example, you might want to activate and deactivate a button elsewhere on the form every time a user enters and exits a specific column.

The following table lists the grid events available in the **Object Inspector**.

Grid control events

Event	Purpose
OnCellClick	Occurs when a user clicks on a cell in the grid.
OnColEnter	Occurs when a user moves into a column on the grid.
OnColExit	Occurs when a user leaves a column on the grid.
OnColumnMoved	Occurs when the user moves a column to a new location.

OnDblClick	Occurs when a user double clicks in the grid.
OnDragDrop	Occurs when a user drags and drops in the grid.
OnDragOver	Occurs when a user drags over the grid.
OnDrawColumnCell	Occurs when application needs to draw individual cells.
OnDrawDataCell	(obsolete) Occurs when application needs to draw individual cells if *State* is *csDefault*.
OnEditButtonClick	Occurs when the user clicks on an ellipsis button in a column.
OnEndDrag	Occurs when a user stops dragging on the grid.
OnEnter	Occurs when the grid gets focus.
OnExit	Occurs when the grid loses focus.
OnKeyDown	Occurs when a user presses any key or key combination on the keyboard when in the grid.
OnKeyPress	Occurs when a user presses a single alphanumeric key on the keyboard when in the grid.
OnKeyUp	Occurs when a user releases a key when in the grid.
OnStartDrag	Occurs when a user starts dragging on the grid.
OnTitleClick	Occurs when a user clicks the title for a column.

There are many uses for these events. For example, you might write a handler for the OnDblClick event that pops up a list from which a user can choose a value to enter in a column. Such a handler would use the SelectedField property to determine to current row and column.

See Also

Using a Grid Control in Its Default State (see page 1765)

Creating a Customized Grid (see page 1754)

Creating Persistent Columns (see page 1756)

Setting Grid Options (see page 1764)

Editing in the Grid (see page 1760)

Displaying ADT and Array Fields (see page 1758)

Controlling Grid Drawing (see page 1753)

3.2.3.9.4.15 Restoring Default Values to a Column

At runtime you can test a column's AssignedValues property to determine whether a column property has been explicitly assigned. Values that are not explicitly defined are dynamically based on the associated field or the grid's defaults.

You can undo property changes made to one or more columns. In the Columns editor, select the column or columns to restore, and then select Restore Defaults from the context menu. Restore defaults discards assigned property settings and restores a column's properties to those derived from its underlying field component

At runtime, you can reset all default properties for a single column by calling the column's RestoreDefaults method. You can also reset default properties for all columns in a grid by calling the column list's *RestoreDefaults* method:

```
DBGrid1.Columns.RestoreDefaults;
DBGrid1->Columns->RestoreDefaults();
```

See Also

Creating Persistent Columns (see page 1756)

Deleting Persistent Columns (see page 1757)

Arranging the Order of Persistent Columns (see page 1751)

Defining a Lookup List Column (see page 1757)

Setting Column Properties at Design Time (see page 1763)

Setting Column Properties at Design Time (see page 1763)

3.2.3.9.4.16 Setting Column Properties at Design Time

Column properties determine how data is displayed in the cells of that column. Most column properties obtain their default values from properties associated with another component (called the *default source*) such as a grid or an associated field component.

To set a column's properties, select the column in The Columns editor and set its properties in the **Object Inspector**. The following table summarizes key column properties you can set.

Column properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers the field data in the column. Default source: *TField*.*Alignment*.
ButtonStyle	*cbsAuto*: (default) Displays a drop-down list if the associated field is a lookup field, or if the column's *PickList* property contains data. *cbsEllipsis*: Displays an ellipsis (...) button to the right of the cell. Clicking on the button fires the grid's *OnEditButtonClick* event. *cbsNone*: The column uses only the normal edit control to edit data in the column.
Color	Specifies the background color of the cells of the column. Default source: *TDBGrid*.*Color*. (For text foreground color, see the Font property.)
DropDownRows	The number of lines of text displayed by the drop-down list. Default: 7.
Expanded	Specifies whether the column is expanded. Only applies to columns representing ADT or array fields.
FieldName	Specifies the field name associated with this column. This can be blank.
ReadOnly	*True*: The data in the column cannot be edited by the user. *False*: (default) The data in the column can be edited.
Width	Specifies the width of the column in screen pixels. Default source: *TField*.*DisplayWidth*.
Font	Specifies the font type, size, and color used to draw text in the column. Default source: *TDBGrid*.*Font*.
PickList	Contains a list of values to display in a drop-down list in the column.
Title	Sets properties for the title of the selected column.

The following table summarizes the options you can specify for the Title property.

Expanded TColumn Title properties

Property	Purpose
Alignment	Left justifies (default), right justifies, or centers the caption text in the column title.
Caption	Specifies the text to display in the column title. Default source: *TField*.*DisplayLabel*.
Color	Specifies the background color used to draw the column title cell. Default source: *TDBGrid*.*FixedColor*.
Font	Specifies the font type, size, and color used to draw text in the column title. Default source: *TDBGrid*.*TitleFont*.

See Also

Creating Persistent Columns (see page 1756)

Deleting Persistent Columns (see page 1757)

Arranging the Order of Persistent Columns (see page 1751)

Defining a Lookup List Column (see page 1757)

Putting a Button in a Column (see page 1761)

Restoring Default Values to a Column (see page 1762)

3.2.3.9.4.17 Setting Grid Options

You can use the grid Options property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the *Options* property in the **Object Inspector** is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of Boolean properties that you can set individually. To view and set these properties, click on the + sign. The list of options in the **Object Inspector** below the *Options* property. The + sign changes to a –(minus) sign, that collapses the list back when you click it.

The following table lists the *Options* properties that can be set, and describes how they affect the grid at runtime.

Expanded TDBGrid Options properties

Option	Purpose
dgEditing	*True*: (Default). Enables editing, inserting, and deleting records in the grid. False: Disables editing, inserting, and deleting records in the grid.
dgAlwaysShowEditor	*True*: When a field is selected, it is in Edit state. False: (Default). A field is not automatically in Edit state when selected.
dgTitles	*True*: (Default). Displays field names across the top of the grid. False: Field name display is turned off.
dgIndicator	*True*: (Default). The indicator column is displayed at the left of the grid, and the current record indicator (an arrow at the left of the grid) is activated to show the current record. On insert, the arrow becomes an asterisk. On edit, the arrow becomes an I-beam. False: The indicator column is turned off.
dgColumnResize	*True*: (Default). Columns can be resized by dragging the column rulers in the title area. Resizing changes the corresponding width of the underlying *TField* component. False: Columns cannot be resized in the grid.
dgColLines	*True*: (Default). Displays vertical dividing lines between columns. False: Does not display dividing lines between columns.
dgRowLines	*True*: (Default). Displays horizontal dividing lines between records. False: Does not display dividing lines between records.
dgTabs	*True*: (Default). Enables tabbing between fields in records. False: Tabbing exits the grid control.
dgRowSelect	*True*: The selection bar spans the entire width of the grid. False: (Default). Selecting a field in a record selects only that field.
dgAlwaysShowSelection	*True*: (Default). The selection bar in the grid is always visible, even if another control has focus. False: The selection bar in the grid is only visible when the grid has focus.
dgConfirmDelete	*True*: (Default). Prompt for confirmation to delete records (`Ctrl+Del`). False: Delete records without confirmation.
dgCancelOnExit	*True*: (Default). Cancels a pending insert when focus leaves the grid. This option prevents inadvertent posting of partial or blank records. False: Permits pending inserts.

dgMultiSelect	*True*: Allows user to select noncontiguous rows in the grid using `Ctrl+Shift` or `Shift+ arrow keys`. False: (Default). Does not allow user to multi-select rows.

See Also

Using a Grid Control in Its Default State (see page 1765)

Creating a Customized Grid (see page 1754)

Editing in the Grid (see page 1760)

Displaying ADT and Array Fields (see page 1758)

Controlling Grid Drawing (see page 1753)

Responding to User Actions at Runtime (see page 1761)

3.2.3.9.4.18 Using a Grid Control in Its Default State

The *Stateproperty of the grid's Columns property indicates whether persistent column objects exist for the grid. Columns.State is a runtime-only property that is automatically set for a grid. The default state is csDefault, meaning that persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined primarily by the properties of the fields in the grid's dataset, or, if there are no persistent field components, by a default set of display characteristics.*

When the grid's Columns.State property is csDefault, grid columns are dynamically generated from the visible fields of the dataset and the order of columns in the grid matches the order of fields in the dataset. Every column in the grid is associated with a field component. Property changes to field components immediately show up in the grid.

Using a grid control with dynamically-generated columns is useful for viewing and editing the contents of arbitrary tables selected at runtime. Because the grid's structure is not set, it can change dynamically to accommodate different datasets. A single grid with dynamically-generated columns can display a Paradox table at one moment, then switch to display the results of an SQL query when the grid's DataSource property changes or when the DataSet property of the data source itself is changed.

You can change the appearance of a dynamic column at design time or runtime, but what you are actually modifying are the corresponding properties of the field component displayed in the column. Properties of dynamic columns exist only so long as a column is associated with a particular field in a single dataset. For example, changing the Width property of a column changes the DisplayWidth property of the field associated with that column. Changes made to column properties that are not based on field properties, such as Font, exist only for the lifetime of the column.

If a grid's dataset consists of dynamic field components, the fields are destroyed each time the dataset is closed. When the field components are destroyed, all dynamic columns associated with them are destroyed as well. If a grid's dataset consists of persistent field components, the field components exist even when the dataset is closed, so the columns associated with those fields also retain their properties when the dataset is closed.

Note: *Changing a grid's Columns.State property to csDefault at runtime deletes all column objects in the grid (even persistent columns), and rebuilds dynamic columns based on the visible fields of the grid's dataset.*

See Also

Creating a Customized Grid (see page 1754)

Setting Grid Options (see page 1764)

Editing in the Grid (see page 1760)

Controlling Grid Drawing (see page 1753)

Responding to User Actions at Runtime (see page 1761)

Displaying ADT and Array Fields (see page 1758)

3.2.3.9.4.19 Using a Single Navigator for Multiple Datasets

As with other data-aware controls, a navigator's *DataSource property specifies the data source that links the control to a dataset. By changing a navigator's DataSource property at runtime, a single navigator can provide record navigation and manipulation for multiple datasets.*

Suppose a form contains two edit controls linked to the CustomersTable and OrdersTable datasets through the CustomersSource and OrdersSource data sources respectively. When a user enters the edit control connected to CustomersSource, the navigator should also use CustomersSource, and when the user enters the edit control connected to OrdersSource, the navigator should switch to OrdersSource as well. You can code an OnEnter event handler for one of the edit controls, and then share that event with the other edit control. For example:

```
procedure TForm1.CustomerCompanyEnter(Sender :TObject);
begin
  if Sender = CustomerCompany then
    DBNavigatorAll.DataSource := CustomerCompany.DataSource
  else
    DBNavigatorAll.DataSource := OrderNum.DataSource;
end;
void __fastcall TForm1::CustomerCompanyEnter(TObject *Sender)
{
  if (Sender == (TObject *)CustomerCompany)
    DBNavigatorAll->DataSource = CustomerCompany->DataSource;
  else
    DBNavigatorAll->DataSource = OrderNum->DataSource;
}
```

See Also

Choosing Navigator Buttons to Display (see page 1752)

Displaying Fly-over Help (see page 1759)

3.2.3.9.4.20 Viewing and Editing Data with TDBGrid

A TDBGrid control lets you view and edit records in a dataset in a tabular grid format.

Three factors affect the appearance of records displayed in a grid control:

- Existence of persistent column objects defined for the grid using the Columns editor. Persistent column objects provide great flexibility setting grid and data appearance. For information on using persistent columns, see Creating a customized grid (see page 1754).
- Creation of persistent field components for the dataset displayed in the grid. For more information about creating persistent field components using the Fields editor, see Working with field components (see page 1877).
- The dataset's *ObjectView* property setting for grids displaying ADT and array fields. See Displaying ADT and array fields (see page 1758).

A grid control has a *Columns property that is itself a wrapper on a TDBGridColumns object. TDBGridColumns is a collection of TColumn objects representing all of the columns in a grid control. You can use the Columns editor to set up column attributes at design time, or use the Columns property of the grid to access the properties, events, and methods of TDBGridColumns at runtime.*

The following topics describe how to use the TDBGrid component:

- Using a Grid Control in Its Default State (see page 1765)
- Creating a Customized Grid (see page 1754)
- Displaying ADT and Array Fields (see page 1758)
- Setting Grid Options (see page 1764)

- Editing in the Grid (see page 1760)
- Controlling Grid Drawing (see page 1753)
- Responding to User Actions at Runtime (see page 1761)

See Also

Creating a Grid That Contains Other Data-aware Controls (see page 1755)

3.2.3.9.4.21 Navigating and Manipulating Records

TDBNavigator provides users a simple control for navigating through records in a dataset, and for manipulating records. The navigator consists of a series of buttons that enable a user to scroll forward or backward through records one at a time, go to the first record, go to the last record, insert a new record, update an existing record, post data changes, cancel data changes, delete a record, and refresh record display.

The following figure shows the navigator that appears by default when you place it on a form at design time. The navigator consists of a series of buttons that let a user navigate from one record to another in a dataset, and edit, delete, insert, and post records. The *VisibleButtons* property of the navigator enables you to hide or show a subset of these buttons dynamically. See Choosing Navigator Buttons to Display (see page 1752) for more information.

The following table describes the buttons on the navigator.

TDBNavigator buttons

Button	Purpose
First	Calls the dataset's *First* method to set the current record to the first record.
Prior	Calls the dataset's *Prior* method to set the current record to the previous record.
Next	Calls the dataset's *Next* method to set the current record to the next record.
Last	Calls the dataset's *Last* method to set the current record to the last record.
Insert	Calls the dataset's *Insert* method to insert a new record before the current record, and set the dataset in Insert state.
Delete	Deletes the current record. If the *ConfirmDelete* property is *True* it prompts for confirmation before deleting.
Edit	Puts the dataset in Edit state so that the current record can be modified.
Post	Writes changes in the current record to the database.
Cancel	Cancels edits to the current record, and returns the dataset to Browse state.
Refresh	Clears data control display buffers, then refreshes its buffers from the physical table or query. Useful if the underlying data may have been changed by another application.

See Displaying fly-over Help (see page 1759) for information on associating help hints with each button. See Using a Single Navigator for Multiple Datasets (see page 1766) for information about associating a navigator with multiple datasets.

See Also

Using Data Sources (see page 1746)

Using Decision Support Components (see page 1800)

Controls that Represent a Single Field

Viewing and Editing Data with TDBGrid (see page 1766)

Creating a Grid That Contains Other Data-aware Controls (see page 1755)

Using Common Data Control Features (see page 1777)

3.2.3.9.5 Disabling and Enabling Data Display

When your application iterates through a dataset or performs a search, you should temporarily prevent refreshing of the values displayed in data-aware controls each time the current record changes. Preventing refreshing of values speeds the iteration or search and prevents annoying screen-flicker.

DisableControls is a dataset method that disables display for all data-aware controls linked to a dataset. As soon as the iteration or search is over, your application should immediately call the dataset's EnableControls method to re-enable display for the controls.

Usually you disable controls before entering an iterative process. The iterative process itself should take place inside a ***try...finally*** *statement so that you can re-enable controls even if an exception occurs during processing. The* ***finally*** *clause should call EnableControls. The following code illustrates how you might use DisableControls and EnableControls in this manner:*

```
CustTable.DisableControls;
try
CustTable.First; { Go to first record, which sets EOF False }
while not CustTable.EOF do { Cycle until EOF is True }
begin
{ Process each record here }
    .
    .
    .
CustTable.Next; { EOF False on success; EOF True when Next fails on last record }
end;
finally
CustTable.EnableControls;
end;
```

```
CustTable->DisableControls();
try
{
  // cycle through all records of the dataset
  for (CustTable->First(); !CustTable->EOF; CustTable->Next())
  {
  // Process each record here
  .
  .
  .
  }
}
__finally
{
  CustTable->EnableControls();
}
```

See Also

Enabling Mouse (see page 1774)

Enabling Editing in Controls On User Entry (see page 1774)

Editing Data in a Control (see page 1773)

Refreshing Data Display (see page 1776)

Enabling and Disabling the Data Source (see page 1774)

3.2.3.9.6 Displaying and Editing Data in List and Combo Boxes

There are four data controls that provide the user with a set of default data values to choose from at runtime. These are

data-aware versions of standard list and combo box controls:

- TDBListBox, which displays a scrollable list of items from which a user can choose to enter in a data field. A data-aware list box displays a default value for a field in the current record and highlights its corresponding entry in the list. If the current row's field value is not in the list, no value is highlighted in the list box. When a user selects a list item, the corresponding field value is changed in the underlying dataset.
- TDBComboBox, which combines the functionality of a data-aware edit control and a drop-down list. At runtime it can display a drop-down list from which a user can pick from a predefined set of values, and it can permit a user to enter an entirely different value.
- TDBLookupListBox, which behaves like *TDBListBox* except the list of display items is looked up in another dataset.
- TDBLookupComboBox, which behaves like *TDBComboBox* except the list of display items is looked up in another dataset.

The following topics describe these components in more detail:

- Using TDBListBox and TDBComboBox (see page 1778)
- Displaying and Editing Data in Lookup List and Combo Boxes (see page 1769)

Note: At runtime, users can use an incremental search to find list box items. When the control has focus, for example, typing 'ROB' selects the first item in the list box beginning with the letters 'ROB'. Typing an additional 'E' selects the first item starting with 'ROBE', such as 'Robert Johnson'. The search is case-insensitive. Backspace

and Esc cancel the current search string (but leave the selection intact), as does a two second pause between keystrokes.

See Also

Displaying Data as Labels (see page 1772)

Displaying and Editing Fields in an Edit Box (see page 1770)

Displaying and Editing Text in a Memo Control (see page 1771)

Displaying and Editing Text in a Rich Edit Memo Control (see page 1772)

Displaying and Editing Graphics Fields in an Image Control (see page 1771)

Handling Boolean Field Values with Check Boxes (see page 1775)

Restricting Field Values with Radio Controls (see page 1748)

3.2.3.9.7 Displaying and Editing Data in Lookup List and Combo Boxes

Lookup list boxes and lookup combo boxes (TDBLookupListBox and TDBLookupComboBox) present the user with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

For example, consider an order form whose fields are tied to the *OrdersTable*. *OrdersTable* contains a *CustNo* field corresponding to a customer ID, but *OrdersTable* does not have any other customer information. The *CustomersTable*, on the other hand, contains a *CustNo* field corresponding to a customer ID, and also contains additional information, such as the customer's company and mailing address. It would be convenient if the order form enabled a clerk to select a customer by company name instead of customer ID when creating an invoice. A *TDBLookupListBox* that displays all company names in *CustomersTable* enables a user to select the company name from the list, and set the *CustNo* on the order form appropriately.

These lookup controls derive the list of display items from one of two sources:

A lookup field defined for a dataset. To specify list box items using a lookup field, the dataset to which you link the control must already define a lookup field (see page 1868).

To specify the lookup field for the list box items

1. Set the DataSource property of the list box to the data source for the dataset containing the lookup field to use.

2. Choose the lookup field to use from the drop-down list for the DataField property.
3. When you activate a table associated with a lookup control, the control recognizes that its data field is a lookup field, and displays the appropriate values from the lookup.

A secondary data source, data field, and key. If you have not defined a lookup field for a dataset, you can establish a similar relationship using a secondary data source, a field value to search on in the secondary data source, and a field value to return as a list item.

To specify a secondary data source for list box items

1. Set the DataSource property of the list box to the data source for the control.
2. Choose a field into which to insert looked-up values from the drop-down list for the DataField*property. The field you choose cannot be a lookup field.*
3. Set the ListSource property of the list box to the data source for the dataset that contain the field whose values you want to look up.
4. Choose a field to use as a lookup key from the drop-down list for the KeyField property. The drop-down list displays fields for the dataset associated with data source you specified in Step 3. The field you choose need not be part of an index, but if it is, lookup performance is even faster.
5. Choose a field whose values to return from the drop-down list for the ListField property. The drop-down list displays fields for the dataset associated with the data source you specified in Step 3.

When you activate a table associated with a lookup control, the control recognizes that its list items are derived from a secondary source, and displays the appropriate values from that source.

To specify the number of items that appear at one time in a *TDBLookupListBox* control, use the RowCount property. The height of the list box is adjusted to fit this row count exactly.

To specify the number of items that appear in the drop-down list of *TDBLookupComboBox*, use the DropDownRows property instead.

Note: You can also set up a column in a data grid to act as a lookup combo box. For information on how to do this, see Defining a lookup list column (see page 1757).

See Also

Using TDBListBox and TDBComboBox (see page 1778)

3.2.3.9.8 Displaying and Editing Fields in an Edit Box

TDBEdit is a data-aware version of an edit box component. *TDBEdit* displays the current value of a data field to which it is linked and permits it to be edited using standard edit box techniques.

For example, suppose *CustomersSource* is a *TDataSource* component that is active and linked to an open TClientDataSet called *CustomersTable*. You can then place a *TDBEdit* component on a form and set its properties as follows:

- *DataSource*: CustomersSource
- *DataField*: CustNo

The data-aware edit box component immediately displays the value of the current row of the *CustNo* column of the *CustomersTable* dataset, both at design time and at runtime.

See Also

Displaying Data as Labels (see page 1772)

Displaying and Editing Text in a Memo Control (see page 1771)

Displaying and Editing Text in a Rich Edit Memo Control (see page 1772)

Displaying and Editing Graphics Fields in an Image Control (see page 1771)

Displaying and Editing Data in List and Combo Boxes (see page 1768)

Handling Boolean Field Values with Check Boxes (see page 1775)

Restricting Field Values with Radio Controls (see page 1748)

3.2.3.9.9 Displaying and Editing Graphics Fields in an Image Control

TDBImage is a data-aware control that displays graphics contained in BLOB fields.

By default, *TDBImage* permits a user to edit a graphics image by cutting and pasting to and from the Clipboard using the CutToClipboard, CopyToClipboard, and PasteFromClipboard methods. You can, instead, supply your own editing methods attached to the event handlers for the control.

By default, an image control displays as much of a graphic as fits in the control, cropping the image if it is too big. You can set the *Stretch property to True to resize the graphic to fit within an image control as it is resized.*

Because the TDBImage can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes scroll through data records, TDBImage has an AutoDisplay property that controls whether the accessed data should automatically displayed. If you set AutoDisplay to False, TDBImage displays the field name rather than actual data. Double-click inside the control to view the actual data.

See Also

Displaying Data as Labels (see page 1772)

Displaying and Editing Fields in an Edit Box (see page 1770)

Displaying and Editing Text in a Memo Control (see page 1771)

Displaying and Editing Text in a Rich Edit Memo Control (see page 1772)

Displaying and Editing Data in List and Combo Boxes (see page 1768)

Handling Boolean Field Values with Check Boxes (see page 1775)

Restricting Field Values with Radio Controls (see page 1748)

3.2.3.9.10 Displaying and Editing Text in a Memo Control

TDBMemo is a data-aware component—similar to the standard TMemo component—that can display lengthy text data. *TDBMemo* displays multi-line text, and permits a user to enter multi-line text as well. You can use *TDBMemo* controls to display large text fields or text data contained in binary large object (BLOB) fields.

By default, *TDBMemo* permits a user to edit memo text. To prevent editing, set the *ReadOnly property of the memo control to True. To display tabs and permit users to enter them in a memo, set the WantTabs property to True. To limit the number of characters users can enter into the database memo, use the MaxLength property. The default value for MaxLength is 0, meaning that there is no character limit other than that imposed by the operating system.*

Several properties affect how the database memo appears and how text is entered. You can supply scroll bars in the memo with the ScrollBars property. To prevent word wrap, set the WordWrap property to False. The Alignment property determines how the text is aligned within the control. Possible choices are taLeftJustify (the default), taCenter, and taRightJustify. To change the font of the text, use the Font property.

At runtime, users can cut, copy, and paste text to and from a database memo control. You can accomplish the same task programmatically by using the CutToClipboard, CopyToClipboard, and PasteFromClipboard methods.

Because the TDBMemo can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, TDBMemo has an AutoDisplay property that controls whether the accessed data should be displayed automatically. If you set AutoDisplay to False, TDBMemo displays the field name rather than actual data. Double-click inside the control to view the actual data.

See Also

Displaying Data as Labels (see page 1772)

Displaying and Editing Fields in an Edit Box (see page 1770)

Displaying and Editing Text in a Rich Edit Memo Control (see page 1772)

Displaying and Editing Graphics Fields in an Image Control (see page 1771)

Displaying and Editing Data in List and Combo Boxes (see page 1768)

Handling Boolean Field Values with Check Boxes (see page 1775)

Restricting Field Values with Radio Controls (see page 1748)

3.2.3.9.11 Displaying and Editing Text in a Rich Edit Memo Control

TDBRichEdit is a data-aware component—similar to the standard *TRichEdit* component—that can display formatted text stored in a binary large object (BLOB) field. *TDBRichEdit* displays formatted, multi-line text, and permits a user to enter formatted multi-line text as well.

Note: While *TDBRichEdit* provides properties and methods to enter and work with rich text, it does not provide any user interface components to make these formatting options available to the user. Your application must implement the user interface to surface rich text capabilities.

By default, *TDBRichEdit* permits a user to edit memo text. To prevent editing, set the *ReadOnly property of the rich edit control to True. To display tabs and permit users to enter them in a memo, set the WantTabs property to True. To limit the number of characters users can enter into the database memo, use the MaxLength property. The default value for MaxLength is 0, meaning that there is no character limit other than that imposed by the operating system.*

Because the TDBRichEdit can display large amounts of data, it can take time to populate the display at runtime. To reduce the time it takes to scroll through data records, TDBRichEdit has an AutoDisplay property that controls whether the accessed data should be displayed automatically. If you set AutoDisplay to False, TDBRichEdit displays the field name rather than actual data. Double-click inside the control to view the actual data.

See Also

Displaying Data as Labels (see page 1772)

Displaying and Editing Fields in an Edit Box (see page 1770)

Displaying and Editing Text in a Memo Control (see page 1771)

Displaying and Editing Graphics Fields in an Image Control (see page 1771)

Displaying and Editing Data in List and Combo Boxes (see page 1768)

Handling Boolean Field Values with Check Boxes (see page 1775)

Restricting Field Values with Radio Controls (see page 1748)

3.2.3.9.12 Displaying Data as Labels

TDBText is a read-only control similar to the TLabel component on the Standard category of the **Tool palette**. A *TDBText* control is useful when you want to provide display-only data on a form that allows user input in other controls. For example, suppose a form is created around the fields in a customer list table, and that once the user enters a street address, city, and state or province information in the form, you use a dynamic lookup to automatically determine the zip code field from a separate table. A *TDBText* component tied to the zip code table could be used to display the zip code field that matches the address entered by

the user.

TDBText gets the text it displays from a specified field in the current record of a dataset. Because *TDBText* gets its text from a dataset, the text it displays is dynamic—the text changes as the user navigates the database table. Therefore you cannot specify the display text of *TDBText* at design time as you can with *TLabel*.

Note: When you place a *TDBText* component on a form, make sure its AutoSize property is *True* (the default) to ensure that the control resizes itself as necessary to display data of varying widths. If *AutoSize* is *False*, and the control is too small, data display is clipped.

See Also

Displaying and Editing Fields in an Edit Box (see page 1770)

Displaying and Editing Text in a Memo Control (see page 1771)

Displaying and Editing Text in a Rich Edit Memo Control (see page 1772)

Displaying and Editing Graphics Fields in an Image Control (see page 1771)

Displaying and Editing Data in List and Combo Boxes (see page 1768)

Handling Boolean Field Values with Check Boxes (see page 1775)

Restricting Field Values with Radio Controls (see page 1748)

3.2.3.9.13 Editing and Updating Data

All data controls except the navigator display data from a database field. In addition, you can use them to edit and update data as long as the underlying dataset allows it.

Note: Unidirectional datasets never permit users to edit and update data.

The following topics describe how to allow users to edit data using data controls:

- Enabling Editing in Controls On User Entry (see page 1774)
- Editing Data in a Control (see page 1773)

See Also

Associating a Data Control with a Dataset (see page 1746)

Disabling and Enabling Data Display (see page 1768)

Refreshing Data Display (see page 1776)

Enabling Mouse (see page 1774)

3.2.3.9.14 Editing Data in a Control

A data control can only post edits to its associated dataset if the dataset's CanModify property is *True*. *CanModify* is always *False* for unidirectional datasets. Some datasets have a *ReadOnly* property that lets you specify whether *CanModify* is *True*.

Note: Whether a dataset can update data depends on whether the underlying database table permits updates.

Even if the dataset's *CanModify* property is *True*, the Enabled property of the data source that connects the dataset to the control must be *True* as well before the control can post updates back to the database table. The *Enabled* property of the data source determines whether the control can display field values from the dataset, and therefore also whether a user can edit and post values. If *Enabled* is *True* (the default), controls can display field values.

Finally, you can control whether the user can even enter edits to the data that is displayed in the control. The *ReadOnly* property

of the data control determines if a user can edit the data displayed by the control. If *False* (the default), users can edit data. Clearly, you will want to ensure that the control's *ReadOnly* property is *True* when the dataset's *CanModify* property is *False*. Otherwise, you give users the false impression that they can affect the data in the underlying database table.

In all data controls except TDBGrid, when you modify a field, the modification is copied to the underlying dataset when you `Tab` from the control. If you press `Esc` before you `Tab` from a field, the data control abandons the modifications, and the value of the field reverts to the value it held before any modifications were made.

In TDBGrid, modifications are posted when you move to a different record; you can press `Esc` in any record of a field before moving to another record to cancel all changes to the record.

When a record is posted, Delphi checks all data-aware controls associated with the dataset for a change in status. If there is a problem updating any fields that contain modified data, Delphi raises an exception, and no modifications are made to the record.

Note: If your application caches updates (for example, using a client dataset), all modifications are posted to an internal cache. These modifications are not applied to the underlying database table until you call the dataset's *ApplyUpdates* method.

See Also

Enabling Editing in Controls On User Entry (see page 1774)

3.2.3.9.15 Enabling and Disabling the Data Source

The data source has an Enabled property that determines if it is connected to its dataset. When *Enabled* is *True*, the data source is connected to a dataset.

You can temporarily disconnect a single data source from its dataset by setting *Enabled* to *False*. When *Enabled* is *False*, all data controls attached to the data source component go blank and become inactive until *Enabled* is set to *True*. It is recommended, however, to control access to a dataset through a dataset component's DisableControls and EnableControls methods because they affect all attached data sources.

See Also

Changing the Associated Dataset at Runtime (see page 1747)

Responding to Changes Mediated by the Data Source (see page 1776)

3.2.3.9.16 Enabling Editing in Controls On User Entry

A dataset must be in *dsEdit* state to permit editing to its data. If the data source's AutoEdit property is *True* (the default), the data control handles the task of putting the dataset into *dsEdit* mode as soon as the user tries to edit its data.

If *AutoEdit* is *False*, you must provide an alternate mechanism for putting the dataset into edit mode. One such mechanism is to use a TDBNavigator control with an *Edit* button, which lets users explicitly put the dataset into edit mode. For more information about *TDBNavigator*, see Navigating and manipulating records (see page 1767). Alternately, you can write code that calls the dataset's Edit method when you want to put the dataset into edit mode.

See Also

Editing Data in a Control (see page 1773)

Enabling and Disabling the Data Source (see page 1774)

3.2.3.9.17 Enabling Mouse, Keyboard, and Timer Events

The Enabled property of a data control determines whether it responds to mouse, keyboard, or timer events, and passes information to its data source. The default setting for this property is *True*.

To prevent mouse, keyboard, or timer events from reaching a data control, set its *Enabled* property to *False*. When *Enabled* is *False*, the data source that connects the control to its dataset does not receive information from the data control. The data control continues to display data, but the text displayed in the control is dimmed.

See Also

Associating a Data Control with a Dataset (see page 1746)

Disabling and Enabling Data Display (see page 1768)

Refreshing Data Display (see page 1776)

Editing and Updating Data (see page 1773)

3.2.3.9.18 Handling Boolean Field Values with Check Boxes

TDBCheckBox is a data-aware check box control. It can be used to set the values of Boolean fields in a dataset. For example, a customer invoice form might have a check box control that when checked indicates the customer is tax-exempt, and when unchecked indicates that the customer is not tax-exempt.

The data-aware check box control manages its checked or unchecked state by comparing the value of the current field to the contents of ValueChecked and ValueUnchecked properties. If the field value matches the *ValueChecked* property, the control is checked. Otherwise, if the field matches the *ValueUnchecked* property, the control is unchecked.

Note: The values in *ValueChecked* and *ValueUnchecked* cannot be identical.

Set the *ValueChecked* property to a value the control should post to the database if the control is checked when the user moves to another record. By default, this value is set to "true," but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueChecked*. If any of the items matches the contents of that field in the current record, the check box is checked. For example, you can specify a *ValueChecked* string like:

```
DBCheckBox1.ValueChecked := 'True;Yes;On';
DBCheckBox1->ValueChecked = "true;Yes;On";
```

If the field for the current record contains values of "True," "Yes," or "On," then the check box is checked. Comparison of the field to *ValueChecked* strings is case-insensitive. If a user checks a box for which there are multiple *ValueChecked* strings, the first string is the value that is posted to the database.

Set the *ValueUnchecked* property to a value the control should post to the database if the control is not checked when the user moves to another record. By default, this value is set to "false," but you can make it any alphanumeric value appropriate to your needs. You can also enter a semicolon-delimited list of items as the value of *ValueUnchecked*. If any of the items matches the contents of that field in the current record, the check box is unchecked.

A data-aware check box is disabled whenever the field for the current record does not contain one of the values listed in the *ValueChecked* or *ValueUnchecked* properties.

If the field with which a check box is associated is a logical field, the check box is always checked if the contents of the field is *True*, and it is unchecked if the contents of the field is *False*. In this case, strings entered in the *ValueChecked* and *ValueUnchecked* properties have no effect on logical fields.

See Also

Displaying Data as Labels (see page 1772)

Displaying and Editing Fields in an Edit Box (see page 1770)

Displaying and Editing Text in a Memo Control (see page 1771)

Displaying and Editing Text in a Rich Edit Memo Control (see page 1772)

Displaying and Editing Graphics Fields in an Image Control (see page 1771)

Displaying and Editing Data in List and Combo Boxes (see page 1768)

Restricting Field Values with Radio Controls (see page 1748)

3.2.3.9.19 Refreshing Data Display

The Refresh method for a dataset flushes local buffers and re-fetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application. If you are using cached updates, before you refresh the dataset you must apply any updates the dataset has currently cached.

Refreshing can sometimes lead to unexpected results. For example, if a user is viewing a record deleted by another application, then the record disappears the moment your application calls *Refresh*. Data can also appear to change if another user changes a record after you originally fetched the data and before you call *Refresh*.

See Also

Associating a Data Control with a Dataset (see page 1746)

Disabling and Enabling Data Display (see page 1768)

Editing and Updating Data (see page 1773)

Enabling Mouse (see page 1774)

3.2.3.9.20 Responding to Changes Mediated by the Data Source

Because the data source provides the link between the data control and its dataset, it mediates all of the communication that occurs between the two. Typically, the data-aware control automatically responds to changes in the dataset. However, if your user interface is using controls that are not data-aware, you can use the events of a data source component to manually provide the same sort of response.

The OnDataChange event occurs whenever the data in a record may have changed, including field edits or when the cursor moves to a new record. This event is useful for making sure the control reflects the current field values in the dataset, because it is triggered by all changes. Typically, an *OnDataChange* event handler refreshes the value of a non-data-aware control that displays field data.

The UpdateData event occurs when the data in the current record is about to be posted. For instance, an *OnUpdateData* event occurs after Post is called, but before the data is actually posted to the underlying database server or local cache.

The StateChange event occurs when the state of the dataset changes. When this event occurs, you can examine the dataset's State property to determine its current state.

For example, the following *OnStateChange* event handler enables or disables buttons or menu items based on the current state:

```
procedure Form1.DataSource1.StateChange(Sender: TObject);
begin
  CustTableEditBtn.Enabled := (CustTable.State = dsBrowse);
  CustTableCancelBtn.Enabled := CustTable.State in [dsInsert, dsEdit, dsSetKey];
  CustTableActivateBtn.Enabled := CustTable.State in [dsInactive];
  .
  .
  .
end;
void __fastcall TForm1::DataSource1StateChange(TObject *Sender)
{
  CustTableActivateBtn->Enabled = (CustTable->State == dsInactive);
  CustTableEditBtn->Enabled = (CustTable->State == dsBrowse);
  CustTableCancelBtn->Enabled = (CustTable->State == dsInsert ||
                                 CustTable->State == dsEdit ||
```

```
                                    CustTable->State == dsSetKey);
    .
    .
    .
}
```

Note: For more information about dataset states, see Determining Dataset States (see page 1633).

See Also

Changing the Associated Dataset at Runtime (see page 1747)

Enabling and Disabling the Data Source (see page 1774)

3.2.3.9.21 Using Common Data Control Features

The following tasks are common to most data controls:

- Associating a data control with a dataset (see page 1746)
- Editing and updating data (see page 1773)
- Disabling and enabling data display (see page 1768)
- Refreshing data display (see page 1776)
- Enabling mouse (see page 1774)

Data controls let you display and edit fields of data associated with the current record in a dataset. The following table summarizes the data controls that appear on the Data Controls category of the **Tool palette**.

Data controls

Data control	Description
TDBGrid	Displays information from a data source in a tabular format. Columns in the grid correspond to columns in the underlying table or query's dataset. Rows in the grid correspond to records.
TDBNavigator	Navigates through data records in a dataset. updating records, posting records, deleting records, canceling edits to records, and refreshing data display.
TDBText	Displays data from a field as a label.
TDBEdit	Displays data from a field in an edit box.
TDBMemo	Displays data from a memo or BLOB field in a scrollable, multi-line edit box.
TDBImage	Displays graphics from a data field in a graphics box.
TDBListBox	Displays a list of items from which to update a field in the current data record.
TDBComboBox	Displays a list of items from which to update a field, and also permits direct text entry like a standard data-aware edit box.
TDBCheckBox	Displays a check box that indicates the value of a Boolean field.
TDBRadioGroup	Displays a set of mutually exclusive options for a field.
TDBLookupListBox	Displays a list of items looked up from another dataset based on the value of a field.
TDBLookupComboBox	Displays a list of items looked up from another dataset based on the value of a field, and also permits direct text entry like a standard data-aware edit box.
TDBCtrlGrid	Displays a configurable, repeating set of data-aware controls within a grid.
TDBRichEdit	Displays formatted data from a field in an edit box.

Data controls are data-aware at design time. When you associate the data control with an active dataset (see page 1746) while building an application, you can immediately see live data in the control. You can use the Fields editor to scroll through a dataset at design time to verify that your application displays data correctly without having to compile and run the application. For

more information about the Fields editor, see Creating Persistent Fields (see page 1874).

At runtime, data controls display data and, if your application, the control, and the dataset all permit it, a user can edit data through the control.

See Also

Choosing How to Organize the Data (see page 1747)

Navigating and Manipulating Records (see page 1767)

3.2.3.9.22 Using TDBListBox and TDBComboBox

When using *TDBListBox* or *TDBComboBox*, you must use the String List editor at design time to create the list of items to display. To bring up the String List editor, click the ellipsis button for the *Items property in the* ***Object Inspector****. Then type in the items that you want to have appear in the list. At runtime, use the methods of the Items property to manipulate its string list.*

When a TDBListBox or TDBComboBox control is linked to a field through its DataField property, the field value appears selected in the list. If the current value is not in the list, no item appears selected. However, TDBComboBox displays the current value for the field in its edit box, regardless of whether it appears in the Items list.

For TDBListBox, the Height property determines how many items are visible in the list box at one time. The IntegralHeight property controls how the last item can appear. If IntegralHeight is False (the default), the bottom of the list box is determined by the ItemHeight property, and the bottom item may not be completely displayed. If IntegralHeight is True, the visible bottom item in the list box is fully displayed.

For TDBComboBox, the Style property determines user interaction with the control. By default, Style is csDropDown, meaning a user can enter values from the keyboard, or choose an item from the drop-down list. The following properties determine how the Items list is displayed at runtime:

- *Style* determines the display style of the component:
- *csDropDown* (default): Displays a drop-down list with an edit box in which the user can enter text. All items are strings and have the same height.
- *csSimple*: Combines an edit control with a fixed size list of items that is always displayed. When setting *Style* to *csSimple*, be sure to increase the *Height* property so that the list is displayed.
- *csDropDownList*: Displays a drop-down list and edit box, but the user cannot enter or change values that are not in the drop-down list at runtime.
- *csOwnerDrawFixed* and *csOwnerDrawVariable*: Allows the items list to display values other than strings (for example, bitmaps) or to use different fonts for individual items in the list.
- DropDownCount: the maximum number of items displayed in the list. If the number of Items is greater than *DropDownCount*, the user can scroll the list. If the number of Items is less than *DropDownCount*, the list will be just large enough to display all the Items.
- ItemHeight: The height of each item when style is *csOwnerDrawFixed*.
- Sorted: If *True*, then the *Items* list is displayed in alphabetical order.

See Also

Displaying and Editing Data in Lookup List and Combo Boxes (see page 1769)

3.2.3.9.23 Using Data Controls

The Data Controls category of the **Tool palette** provides a set of data-aware controls that represent data from fields in a database record, and, if the dataset allows it, enable users to edit that data and post changes back to the database. By placing data controls onto the forms in your database application, you can build your database application's user interface (UI) so that information is visible and accessible to users.

The data-aware controls you add to your user interface depend on several factors, including the following:

- The type of data you are displaying. You can choose between controls that are designed to display and edit plain text, controls that work with formatted text, controls for graphics, multimedia elements, and so on. Controls that display different types of information are described in Displaying a Single Record.
- How you want to organize the information. You may choose to display information from a single record on the screen, or list the information from multiple records using a grid. Choosing how to organize the data describes some of the possibilities.
- The type of dataset that supplies data to the controls. You want to use controls that reflect the limitations of the underlying dataset. For example, you would not use a grid with a unidirectional dataset because unidirectional datasets can only supply a single record at a time.
- How (or if) you want to let users navigate through the records of datasets and add or edit data. You may want to add your own controls or mechanisms to navigate and edit, or you may want to use a built-in control such as a data navigator (see page 1767).

  Note: More complex data-aware controls for decision support are discussed in Using Decision Support Components (see page 1800).

  Regardless of the data-aware controls you choose to add to your interface, certain common features apply. These are described in Using Common Data Control Features (see page 1777).

See Also

Designing Database Applications (see page 1566)

Understanding Datasets (see page 1632)

Working with Field Components (see page 1877)

Creating and Using a Client Dataset (see page 1740)

Connecting to Databases (see page 1506)

Working with ADO Components (see page 1494)

Using Unidirectional Datasets (see page 1823)

Using the Borland Database Engine (see page 1643)

Using Provider Components (see page 1819)

Creating Multi-tiered Applications: Overview (see page 1518)

Using Decision Support Components (see page 1800)

Using XML in Database Applications (see page 1847)

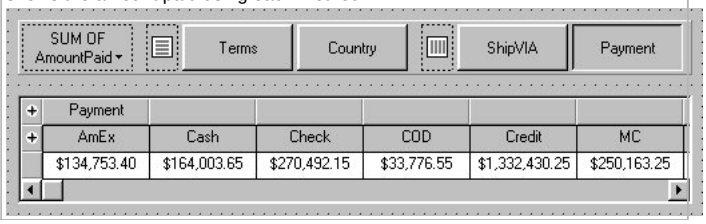
3.2.3.10 Using decision support components

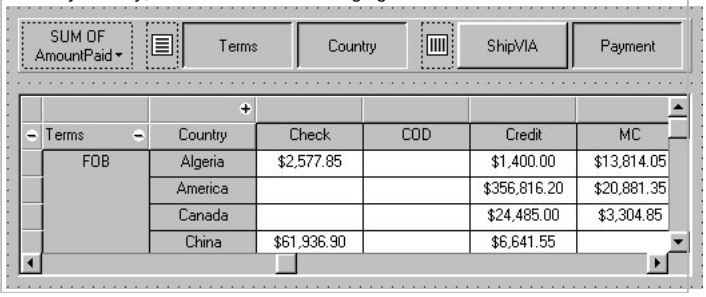
Topics

Name	Description
Changing Other Decision Graph Series Properties (see page 1786)	
Changing Other Decision Graph Template Properties (see page 1786)	
Changing the Default Decision Graph Type (see page 1786)	
Changing the Series Graph Type (see page 1787)	By default, each series has the same graph type, defined by the template for its dimension. To change all series to the same graph type, you can change the template type. See Changing other decision graph series properties. (see page 1786) for instructions.
Creating Decision Datasets with TQuery or TTable (see page 1787)	If you use an ordinary TQuery component as a decision dataset, you must manually set up the SQL statement, taking care to supply a GROUP BY phrase which contains the same fields (and in the same order) as the SELECT phrase. The SQL should look similar to this:

Creating Decision Grids (see page 1788)	
Decision Cube Properties and Events (see page 1788)	The DimensionMap properties of *TDecisionCube* not only control which dimensions and summaries appear but also let you set date ranges and specify the maximum number of dimensions the decision cube may support. You can also indicate whether or not to display data during design. You can display names, (categories) values, subtotals, or data. Display of data at design time can be time consuming, depending on the data source. When you click the ellipsis next to *DimensionMap* in the **Object Inspector**, the Decision Cube editor dialog box appears. You can use its pages and controls to set the *DimensionMap* properties.... more (see page 1788)
Decision Graphs at Runtime (see page 1788)	Users can drag from side to side or up and down in the graph grid area to scroll through off-screen categories and values.
Decision Grids at Runtime (see page 1788)	Users can: Right-click within the decision grid and choose to: • Toggle subtotals on and off for individual data groups, for all values of a dimension, or for the whole grid. • Display the Decision Cube editor, described in Using the Decision Cube editor. (see page 1796) • Toggle dimensions and summaries open and closed. Click + and – within the row and column headings to open and close dimensions. Drag and drop dimensions from rows to columns and the reverse.
Decision Pivots: Runtime Behavior (see page 1789)	Users can: Left-click the summary button at the left end of the decision pivot to display a list of available summaries. They can use this list to change the summary data displayed in decision grids and decision graphs. Right-click a dimension button and choose to: • Move it from the row area to the column area or the reverse. • Drill In to display detail data. Left-click a dimension button following the Drill In command and choose: • Open Dimension to move back to the top level of that dimension. • All Values to toggle between displaying just summaries and summaries plus all other... more (see page 1789)
Decision Support Components at Runtime (see page 1789)	At runtime, users can perform many operations by left-clicking, right-clicking, and dragging visible decision support components. These operations are summarized below. • Decision Pivots at Runtime (see page 1789) • Decision Grids at Runtime (see page 1788) • Decision Graphs at Runtime (see page 1788)
Drilling Down for Detail in Decision Grids (see page 1790)	You can drill down to see more detail in a dimension. For example, if you right-click a category label (row heading) for a dimension with others collapsed beneath it, you can choose to drill down and only see data for that category. When a dimension is drilled, you do not see the category labels for that dimension displayed on the grid, since only the records for a single category value are being displayed. If you have a decision pivot on the form, it displays category values and lets you change to other values if you want. To drill down into... more (see page 1790)
Limiting Dimension Selection in Decision Grids (see page 1790)	You can change the *ControlType* property of the decision source to determine whether more than one dimension can be selected for each axis of the grid. For more information, see Using Decision Sources. (see page 1803)

Opening and Closing Decision Grid Fields (see page 1790)	If a plus sign (+) appears in a dimension or summary field, one or more fields to its right are closed (hidden). You can open additional fields and categories by clicking the sign. A minus sign (–) indicates a fully opened (expanded) field. When you click the sign, the field closes. This outlining feature can be disabled; see Decision Grid Properties (see page 1798) for details.
Overview of Decision Support Components (see page 1790)	The decision support components appear on the Decision Cube category of the **Tool Palette**: • The decision cube, TDecisionCube, is a multidimensional data store. For more information see Using decision cubes. (see page 1797) • The decision source, TDecisionSource, defines the current pivot state of a decision grid or a decision graph. For more information, see Using decision sources. (see page 1803) • The decision query, TDecisionQuery, is a specialized form of TQuery used to define the data in a decision cube. For more information, see Using datasets with decision support components. (see page 1799) • The decision pivot, TDecisionPivot, lets you open or close decision cube dimensions, or fields, by pressing... more (see page 1790)
Decision Pivot Properties (see page 1791)	The following are some special properties that control the appearance and behavior of decision pivots: • The first properties listed for *TDecisionPivot* define its overall behavior and appearance. You might want to set *ButtonAutoSize* to *False* for *TDecisionPivot* to keep buttons from expanding and contracting as you adjust the size of the component. • The *Groups* property of *TDecisionPivot* defines which dimension buttons appear. You can display the row, column, and summary selection button groups in any combination. Note that if you want more flexibility over the placement of these groups, you can place one *TDecisionPivot* on your form which contains only... more (see page 1791)
Reorganizing Rows and Columns in Decision Grids (see page 1792)	You can drag row and column headings to new locations within the same axis or to the other axis. In this way, you can reorganize the grid and see the data from new perspectives as the data groupings change. This pivoting feature can be disabled; see Decision Grid Properties (see page 1798) for details. If you included a decision pivot, you can push and drag its buttons to reorganize the display. See Using decision Pivots (see page 1798) for instructions.
Saving Decision Graph Series Settings (see page 1792)	By default, only settings for templates are saved at design time. Changes made to specific series are only saved if the Save box is checked for that series in the **Chart Editing** dialog box. Saving series can be memory intensive, so if you don't need to save them you can uncheck the Save box.
Setting Decision Graph Template Defaults (see page 1792)	Decision graphs display the values from two dimensions of the decision cube: one dimension is displayed as an axis of the graph, and the other is used to create a set of series. The template for that dimension provides default properties for those series (such as whether the series are bar, line, area, and so on). As users pivot from one state to another, any required series for the dimension are created using the series type and other defaults specified in the template. A separate template is provided for cases where users pivot to a state where only one dimension... more (see page 1792)

Setting Dimension State (see page 1792)	The *ActiveFlag* property controls which dimensions get loaded. You can set this property on the Dimension Settings tab of the Decision Cube editor using the Activity Type control. When this control is set to *Active*, the dimension is loaded unconditionally, and will always take up space. Note that the number of dimensions in this state must always be less than *MaxDimensions*, and the number of summaries set to *Active* must be less than *MaxSummaries*. You should set a dimension or summary to *Active* only when it is critical that it be available at all times. An *Active... more (see page 1792)*
Setting Maximum Dimensions, Summaries, and Cells (see page 1793)	The decision cube's *MaxDimensions* and *MaxSummaries* properties can be used with the *CubeDim.ActiveFlag* property to control how many dimensions and summaries can be loaded at a time. You can set the maximum values on the Cube Capacity page of the Decision Cube editor to place some overall control on how many dimensions or summaries can be brought into memory at the same time. Limiting the number of dimensions or summaries provides a rough limit on the amount of memory used by the decision cube. However, it does not distinguish between dimensions with many values and those with only a few.... more (see page 1793)
Setting the Maximum Available Dimensions and Summaries (see page 1793)	To determine the maximum number of dimensions and summaries available for decision pivots, decision grids, and decision graphs bound to the selected decision cube, display the Decision Cube editor and click the Memory Control tab. Use the edit controls to adjust the current settings, if necessary. These settings help to control the amount of memory required by the decision cube. For more information, see Decision Support Components and Memory Control. (see page 1800)
Using Paged Dimensions (see page 1793)	When Binning is set to Set on the Dimension Settings tab of the Decision cube editor and Start Value is not NULL, the dimension is said to be "paged," or "permanently drilled down." You can access data for just a single value of that dimension at a time, although you can programmatically access a series of values sequentially. Such a dimension may not be pivoted or opened. It is extremely memory intensive to include dimensional data for dimensions that have very large numbers of values. By making such dimensions paged, you can display summary information for one value at a... more (see page 1793)
Viewing and Changing Design Options (see page 1794)	To determine how much information appears at design time, display the Decision Cube editor and click the Memory Control tab. Then, check the setting that indicates which names and data to display. Display of data or field names at design time can cause performance delays in some cases because of the time needed to fetch the data.
Viewing and Changing Dimension Settings (see page 1794)	To view the dimension settings, display the Decision Cube editor and click the Dimension Settings tab. Then, select a dimension or summary in the Available Fields list. Its information appears in the boxes on the right side of the editor: • To change the dimension or summary name that appears in the decision pivot, decision grid, or decision graph, enter a new name in the Display Name edit box. • To determine whether the selected field is a dimension or summary, read the text in the Type edit box. If the dataset is a *TTable* component, you can use Type to specify... more (see page 1794)
Viewing Overall Decision Graph Properties (see page 1794)	
One-Dimensional Crosstabs (see page 1795)	One-dimensional crosstabs show a summary row (or column) for the categories of a single dimension. For example, if Payment is the chosen column dimension and Amount Paid is the summary category, the crosstab in the following figure shows the amount paid using each method. 

Using Decision Graphs (see page 1795)	The decision graph component, TDecisionGraph, displays fields from the decision source TDecisionSource as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot TDecisionPivot. Graphed data comes from a specially formatted dataset such as TDecisionQuery. For an overview of how the decision support components handle and arrange this data, see Using Decision Support Components. (see page 1800) By default, the first row dimension appears as the x-axis and the first column dimension appears as the y-axis. You can use decision graphs instead of or in addition to decision grids, which present cross-tabulated data... more (see page 1795)
The Decision Graph Display (see page 1796)	By default, the decision graph plots summary values for categories in the first active row field (along the y-axis) against values in the first active column field (along the x-axis). Each graphed category appears as a separate series. If only one dimension is selected—for example, by clicking only one *TDecisionPivot* button—only one series is graphed. If you used a decision pivot, you can push its buttons to determine which decision cube fields (dimensions) are graphed. To exchange graph axes, drag the decision pivot dimension buttons from one side of the separator space to the other. If you have a one-dimensional... more (see page 1796)
Using the Decision Cube Editor (see page 1796)	You can use the Decision Cube editor to set the DimensionMap properties of decision cubes. You can display the Decision Cube editor through the **Object Inspector**, as described in the previous section, or by right-clicking a decision cube on a form at design time and choosing Decision Cube editor. The Decision Cube Editor dialog box has two tabs: • Dimension Settings (see page 1794) used to activate or disable available dimensions, rename and reformat dimensions, put dimensions in a permanently drilled state, and set date ranges to display. • Memory Control, used to set the maximum number of dimensions and summaries (see page 1793) that can be... more (see page 1796)
Creating and Using Decision Graphs (see page 1796)	Decision graph components, TDecisionGraph, present cross-tabulated data in graphic form. Each decision graph shows the value of a single summary, such as Sum, Count, or Avg, charted for one or more dimensions. For more information on crosstabs, see One-dimensional crosstabs. (see page 1795) For illustrations of decision graphs at design time, see the figures Decision support components at design time (see page 1790) and Decision graphs bound to different decision sources. (see page 1795) The following topics are discussed in this section: • Creating Decision Graphs (see page 1802) • Using Decision Graphs (see page 1795) • The Decision Graph Display (see page 1796) • Customizing Decision Graphs (see page 1801)
Multidimensional Crosstabs (see page 1797)	Multidimensional crosstabs use additional dimensions for the rows and/or columns. For example, a two-dimensional crosstab could show amounts paid by payment method for each country. A three-dimensional crosstab could show amounts paid by payment method and terms by country, as shown in the following figure. 

Using Decision Cubes (see page 1797)	The decision cube component, TDecisionCube, is a multidimensional data store that fetches its data from a dataset (typically a specially structured SQL statement entered through *TDecisionQuery* or *TQuery*). The data is stored in a form that makes its easy to pivot (that is, change the way in which the data is organized and summarized) without needing to run the query a second time. The following topics are discussed in this section: • Decision Cube Properties and Events (see page 1788) • Using the Decision Cube Editor (see page 1796)
Decision Grid Properties (see page 1798)	The decision grid component, TDecisionGrid, displays data from the TDecisionSource component bound to TDecisionSource. By default, data appears in a grid with category fields on the left side and top of the grid. The following are some special properties that control the appearance and behavior of decision grids: • *TDecisionGrid* has unique properties for each dimension. To set these, choose *Dimensions* in the **Object Inspector**, then select a dimension. Its properties then appear in the **Object Inspector**: *Alignment* defines the alignment of category labels for that dimension, *Caption* can be used to override the default dimension name, *Color* defines... more (see page 1798)
Using Decision Pivots (see page 1798)	The decision pivot component, TDecisionPivot, lets you open or close decision cube dimensions, or fields, by pressing buttons. When a row or column is opened by pressing a *TDecisionPivot* button, the corresponding dimension appears on the TDecisionGrid or TDecisionGraph component. When a dimension is closed, its detailed data doesn't appear; it collapses into the totals of other dimensions. A dimension may also be in a "drilled" state, where only the summaries for a particular value of the dimension field appear. You can also use the decision pivot to reorganize dimensions displayed in the decision grid and decision graph. Just drag... more (see page 1798)
Customizing Decision Graph Series (see page 1799)	The templates supply many defaults for each decision cube dimension, such as graph type and how series are displayed. Other defaults, such as series color, are defined by *TDecisionGraph*. If you want you can override the defaults for each series. The templates are intended for use when you want the program to create the series for categories as they are needed, and discard them when they are no longer needed. If you want, you can set up custom series for specific category values. To do this, pivot the graph so its current display has a series for the category... more (see page 1799)
Using Datasets with Decision Support Components (see page 1799)	The only decision support component that binds directly to a dataset is the decision cube, TDecisionCube. *TDecisionCube* expects to receive data with groups and summaries defined by an SQL statement of an acceptable format. The GROUP BY phrase must contain the same non-summarized fields (and in the same order) as the SELECT phrase, and summary fields must be identified. The decision query component, TDecisionQuery, is a specialized form of *TQuery*. You can use TDecisionQuery (see page 1802) to more simply define the setup of dimensions (rows and columns) and summary values used to supply data to decision cubes (*TDecisionCube*). The... more (see page 1799)
Using Decision Support Components (see page 1800)	The decision support components help you create cross-tabulated—or, crosstab—tables and graphs. You can then use these tables and graphs to view and summarize data from different perspectives. For more information on cross-tabulated data, see About crosstabs. (see page 1804) The following topics are discussed in this section: • Overview of Decision Support Components (see page 1790) • Guidelines for Using Decision Support Components (see page 1804) • Decision Support Components at Runtime (see page 1789) • Decision Support Components and Memory Control (see page 1800)

Decision Support Components and Memory Control (see page 1800)	When a dimension or summary is loaded into the decision cube, it takes up memory. Adding a new summary increases memory consumption linearly: that is, a decision cube with two summaries uses twice as much memory as the same cube with only one summary, a decision cube with three summaries uses three times as much memory as the same cube with one summary, and so on. Memory consumption for dimensions increases more quickly. Adding a dimension with 10 values increases memory consumption by a factor of 10. Adding a dimension with 100 values increases memory consumption 100 times. Thus adding... more (see page 1800)
Using Decision Grids (see page 1801)	The decision grid component, TDecisionGrid, displays data from decision cubes TDecisionCube bound to decision sources TDecisionSource. By default, the grid appears with dimension fields at its left side and/or top, depending on the grouping instructions defined in the dataset. Categories, one for each data value, appear under each field. You can • Open and Close Dimensions (see page 1790) • Reorganize (see page 1792) • Drill Down for Detail (see page 1790) • Limit Dimension Selection to a Single Dimension for Each Axis (see page 1790) For more information about special properties and events of the decision grid, see Decision grid properties. (see page 1798)
Customizing Decision Graphs (see page 1801)	The decision graph component, TDecisionGraph, displays fields from the decision source (TDecisionSource) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot TDecisionPivot. You can change the type, colors, marker types for line graphs, and many other properties of decision graphs.
Creating Decision Graphs (see page 1802)	
Creating Decision Datasets with the Decision Query Editor (see page 1802)	All data used by the decision support components passes through the decision cube, which accepts a specially formatted dataset most easily produced by an SQL query. See Using datasets with decision support components (see page 1799) for more information. While both *TTable* and *TQuery* can be used as decision datasets, it is easier to use TDecisionQuery; the Decision Query editor supplied with it can be used to specify tables, fields, and summaries to appear in the decision cube and will help you set up the SELECT and GROUP BY portions of the SQL correctly.
Using Decision Sources (see page 1803)	The decision source component, TDecisionSource, defines the current pivot state of decision grids or decision graphs. Any two objects which use the same decision source also share pivot states. The following are some special properties and events that control the appearance and behavior of decision sources: • The ControlType property of *TDecisionSource* indicates whether the decision pivot buttons should act like check boxes (multiple selections) or radio buttons (mutually exclusive selections). • The SparseCols and SparseRows properties of *TDecisionSource* indicate whether to display columns or rows with no values; if *True*, sparse columns or rows are displayed. TDecisionSource has the following... more (see page 1803)
Creating and Using Decision Grids (see page 1803)	Decision grid components, TDecisionGrid, present cross-tabulated data in table form. These tables are also called crosstabs, described in About crosstabs. (see page 1804) The figure Decision support components at design time (see page 1790) shows a decision grid on a form at design time. The following topics are discussed in this section: • Creating Decision Grids (see page 1788) • Using Decision Grids (see page 1801) • Decision Grid Properties (see page 1798)

About Crosstabs (see page 1804)	Cross-tabulations, or crosstabs, are a way of presenting subsets of data so that relationships and trends are more visible. Table fields become the dimensions of the crosstab while field values define categories and summaries within a dimension. You can use the decision support components to set up crosstabs in forms. TDecisionGrid shows data in a table, while TDecisionGraph charts it graphically. TDecisionPivot has buttons that make it easier to display and hide dimensions and move them between columns and rows. Crosstabs can be one-dimensional or multidimensional. The following topics are discussed in this section: • One-Dimensional Crosstabs (see page 1795) • Multidimensional Crosstabs (see page 1797)
Guidelines for Using Decision Support Components (see page 1804)	The decision support components listed in Overview of decision support components (see page 1790) can be used together to present multidimensional data as tables and graphs. More than one grid or graph can be attached to each dataset. More than one instance of *TDecisionPivot* can be used to display the data from different perspectives at runtime.

3.2.3.10.1 Changing Other Decision Graph Series Properties

To change color or other properties of a decision graph series

1. Select the Series page at the top of the Chart Editing dialog box.
2. Choose a series in the drop-down list at the top of the page.
3. Choose the appropriate property tab and select settings.
4. Check the Save Series check box.

See Also

Changing the Series Graph Type (see page 1787)

Saving Decision Graph Series Settings (see page 1792)

3.2.3.10.2 Changing Other Decision Graph Template Properties

To change color or other properties of a template

1. Select the Series page at the top of the Chart Editing dialog box.
2. Choose a template in the drop-down list at the top of the page.
3. Choose the appropriate property tab and select settings.

See Also

Changing the Default Decision Graph Type (see page 1786)

Viewing Overall Decision Graph Properties (see page 1794)

3.2.3.10.3 Changing the Default Decision Graph Type

To change the default graph type

1. Select a template in the Series list on the Chart page of the Chart Editing dialog box.
2. Click the Change button.
3. Select a new type and close the Gallery dialog box.

See Also

Changing Other Decision Graph Template Properties (see page 1786)

Viewing Overall Decision Graph Properties (see page 1794)

3.2.3.10.4 Changing the Series Graph Type

By default, each series has the same graph type, defined by the template for its dimension. To change all series to the same graph type, you can change the template type. See Changing other decision graph series properties. (see page 1786) for instructions.

To change the graph type for a single series

1. Select a series in the Series list on the Chart page of the Chart editor.
2. Click the Change button.
3. Select a new type and close the Gallery dialog box.
4. Check the Save Series check box.

See Also

Changing Other Decision Graph Series Properties (see page 1786)

Saving Decision Graph Series Settings (see page 1792)

3.2.3.10.5 Creating Decision Datasets with TQuery or TTable

If you use an ordinary TQuery component as a decision dataset, you must manually set up the SQL statement, taking care to supply a GROUP BY phrase which contains the same fields (and in the same order) as the SELECT phrase.

The SQL should look similar to this:

```
SELECT ORDERS."Terms", ORDERS."ShipVIA",
  ORDERS."PaymentMethod", SUM( ORDERS."AmountPaid" )
FROM "ORDERS.DB" ORDERS
GROUP BY ORDERS."Terms", ORDERS."ShipVIA", ORDERS."PaymentMethod"
```

The ordering of the SELECT fields should match the ordering of the GROUP BY fields. Queries are described in more detail in Using TQuery (see page 1674).

With TTable, you must supply information to the decision cube about which of the fields in the query are grouping fields, and which are summaries. To do this, Fill in the Dimension Type for each field in the *DimensionMap* of the Decision Cube. You must indicate whether each field is a dimension or a summary, and if a summary, you must provide the summary type. Since pivoting averages depend on SUM/COUNT calculations, you must also provide the base field name to allow the decision cube to match pairs of SUM and COUNT aggregators.

See Also

Creating Decision Datasets with the Decision Query Editor (see page 1802)

Working with Tables (see page 1674)

Working with Queries (see page 1674)

3.2.3.10.6 Creating Decision Grids

To create a form with one or more tables of cross-tabulated data

1. Follow steps 1–3 listed under Guidelines for using decision support components. (see page 1804)
2. Add one or more decision grid components (TDecisionGrid) and bind them to the decision source, TDecisionSource, with the **Object Inspector** by setting their DecisionSource property to the appropriate decision source component.
3. Continue with steps 5–7 listed under Guidelines for using decision support components. (see page 1804)

For a description of what appears in the decision grid and how to use it, see Using decision grids. (see page 1801).

To add a graph to the form, follow the instructions in Creating decision graphs. (see page 1796)

See Also

Decision Grid Properties (see page 1798)

3.2.3.10.7 Decision Cube Properties and Events

The DimensionMap properties of *TDecisionCube* not only control which dimensions and summaries appear but also let you set date ranges and specify the maximum number of dimensions the decision cube may support. You can also indicate whether or not to display data during design. You can display names, (categories) values, subtotals, or data. Display of data at design time can be time consuming, depending on the data source.

When you click the ellipsis next to *DimensionMap* in the **Object Inspector**, the Decision Cube editor dialog box appears. You can use its pages and controls to set the *DimensionMap* properties.

The OnRefresh event fires whenever the decision cube cache is rebuilt. Developers can access the new dimension map and change it at that time to free up memory, change the maximum summaries or dimensions, and so on. *OnRefresh* is also useful if users access the Decision Cube editor; application code can respond to user changes at that time.

See Also

Using the Decision Cube Editor (see page 1796)

3.2.3.10.8 Decision Graphs at Runtime

Users can drag from side to side or up and down in the graph grid area to scroll through off-screen categories and values.

See Also

Decision Pivots: Runtime Behavior (see page 1789)

Decision Grids at Runtime (see page 1788)

Using Decision Graphs (see page 1795)

3.2.3.10.9 Decision Grids at Runtime

Users can:

Right-click within the decision grid and choose to:

- Toggle subtotals on and off for individual data groups, for all values of a dimension, or for the whole grid.
- Display the Decision Cube editor, described in Using the Decision Cube editor. (see page 1796)
- Toggle dimensions and summaries open and closed.

Click + and – within the row and column headings to open and close dimensions.

Drag and drop dimensions from rows to columns and the reverse.

See Also

Decision Pivots: Runtime Behavior (see page 1789)

Decision Graphs at Runtime (see page 1788)

Using Decision Grids (see page 1801)

3.2.3.10.10 Decision Pivots: Runtime Behavior

Users can:

Left-click the summary button at the left end of the decision pivot to display a list of available summaries. They can use this list to change the summary data displayed in decision grids and decision graphs.

Right-click a dimension button and choose to:

- Move it from the row area to the column area or the reverse.
- Drill In to display detail data.

Left-click a dimension button following the Drill In command and choose:

- Open Dimension to move back to the top level of that dimension.
- All Values to toggle between displaying just summaries and summaries plus all other values in decision grids.
- From a list of available categories for that dimension, a category to drill into for detail values.

Left-click a dimension button to open or close that dimension.

Drag and drop dimension buttons from the row area to the column area and the reverse; they can drop them next to existing buttons in that area or onto the row or column icon.

See Also

Decision Grids at Runtime (see page 1788)

Decision Graphs at Runtime (see page 1788)

Using Decision Pivots (see page 1799)

3.2.3.10.11 Decision Support Components at Runtime

At runtime, users can perform many operations by left-clicking, right-clicking, and dragging visible decision support components. These operations are summarized below.

- Decision Pivots at Runtime (see page 1789)
- Decision Grids at Runtime (see page 1788)
- Decision Graphs at Runtime (see page 1788)

See Also

Overview of Decision Support Components (see page 1790)

About Crosstabs (see page 1804)

Guidelines for Using Decision Support Components (see page 1804)

Decision Support Components and Memory Control (see page 1800)

3.2.3.10.12 Drilling Down for Detail in Decision Grids

You can drill down to see more detail in a dimension.

For example, if you right-click a category label (row heading) for a dimension with others collapsed beneath it, you can choose to drill down and only see data for that category. When a dimension is drilled, you do not see the category labels for that dimension displayed on the grid, since only the records for a single category value are being displayed. If you have a decision pivot on the form, it displays category values and lets you change to other values if you want.

To drill down into a dimension,

- Right-click a category label and choose Drill In To This Value, or
- Right-click a pivot button and choose Drilled In.

To make the complete dimension active again,

- Right-click the corresponding pivot button or,
- Right-click the decision grid in the upper-left corner and select the dimension.

See Also

Opening and Closing Decision Grid Fields (see page 1790)

Reorganizing Rows and Columns in Decision Grids (see page 1792)

Limiting Dimension Selection in Decision Grids (see page 1790)

3.2.3.10.13 Limiting Dimension Selection in Decision Grids

You can change the *ControlType* property of the decision source to determine whether more than one dimension can be selected for each axis of the grid. For more information, see Using Decision Sources. (see page 1803)

See Also

Opening and Closing Decision Grid Fields (see page 1790)

Reorganizing Rows and Columns in Decision Grids (see page 1792)

Drilling Down for Detail in Decision Grids (see page 1790)

3.2.3.10.14 Opening and Closing Decision Grid Fields

If a plus sign (+) appears in a dimension or summary field, one or more fields to its right are closed (hidden). You can open additional fields and categories by clicking the sign. A minus sign (–) indicates a fully opened (expanded) field. When you click the sign, the field closes. This outlining feature can be disabled; see Decision Grid Properties (see page 1798) for details.

See Also

Reorganizing Rows and Columns in Decision Grids (see page 1792)

Drilling Down for Detail in Decision Grids (see page 1790)

Limiting Dimension Selection in Decision Grids (see page 1790)

3.2.3.10.15 Overview of Decision Support Components

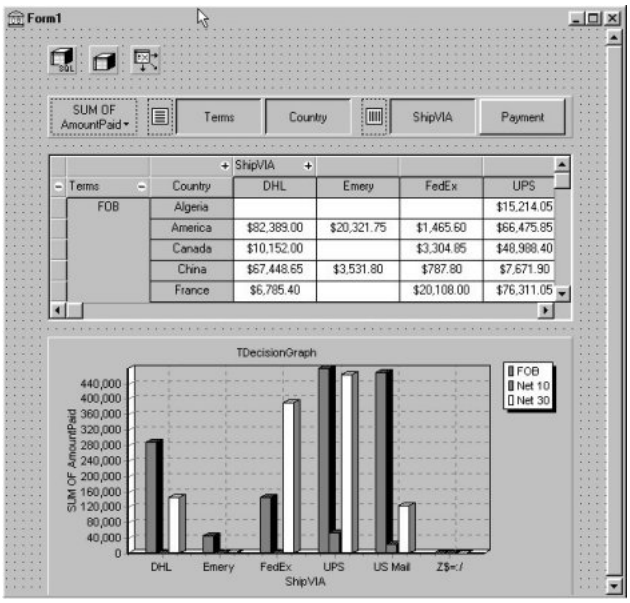
The decision support components appear on the Decision Cube category of the **Tool Palette**:

- The decision cube, TDecisionCube, is a multidimensional data store. For more information see Using decision cubes. (see

page 1797)

- The decision source, TDecisionSource, defines the current pivot state of a decision grid or a decision graph. For more information, see Using decision sources. (see page 1803)
- The decision query, TDecisionQuery, is a specialized form of TQuery used to define the data in a decision cube. For more information, see Using datasets with decision support components. (see page 1799)
- The decision pivot, TDecisionPivot, lets you open or close decision cube dimensions, or fields, by pressing buttons. For more information, see Using decision pivots. (see page 1798)
- The decision grid, TDecisionGrid, displays single- and multidimensional data in table form. For more information, see Creating and using decision grids. (see page 1803)
- The decision graph, TDecisionGraph, displays fields from a decision grid as a dynamic graph that changes when data dimensions are modified. For more information, see Creating and using decision graphs. (see page 1796)

The following figure shows all the decision support components placed on a form at design time.



See Also

Guidelines for Using Decision Support Components (see page 1804)

Decision Support Components and Memory Control (see page 1800)

3.2.3.10.16 Decision Pivot Properties

The following are some special properties that control the appearance and behavior of decision pivots:

- The first properties listed for *TDecisionPivot* define its overall behavior and appearance. You might want to set *ButtonAutoSize* to *False* for *TDecisionPivot* to keep buttons from expanding and contracting as you adjust the size of the component.
- The *Groups* property of *TDecisionPivot* defines which dimension buttons appear. You can display the row, column, and summary selection button groups in any combination. Note that if you want more flexibility over the placement of these groups, you can place one *TDecisionPivot* on your form which contains only rows in one location, and a second which contains only columns in another location.
- Typically, *TDecisionPivot* is added above *TDecisionGrid*. In its default orientation, horizontal, buttons on the left side of *TDecisionPivot* apply to fields on the left side of *TDecisionGrid* (rows); buttons on the right side apply to fields at the top of *TDecisionGrid* (columns).
- You can determine where *TDecisionPivot*'s buttons appear by setting its GroupLayout property to *xtVertical*, *xtLeftTop*, or

xtHorizontal (the default, described in the previous paragraph).

3.2.3.10.17 Reorganizing Rows and Columns in Decision Grids

You can drag row and column headings to new locations within the same axis or to the other axis. In this way, you can reorganize the grid and see the data from new perspectives as the data groupings change. This pivoting feature can be disabled; see Decision Grid Properties (see page 1798) for details.

If you included a decision pivot, you can push and drag its buttons to reorganize the display. See Using decision Pivots (see page 1798) for instructions.

See Also

Opening and Closing Decision Grid Fields (see page 1790)

Drilling Down for Detail in Decision Grids (see page 1790)

Limiting Dimension Selection in Decision Grids (see page 1790)

3.2.3.10.18 Saving Decision Graph Series Settings

By default, only settings for templates are saved at design time. Changes made to specific series are only saved if the Save box is checked for that series in the **Chart Editing** dialog box.

Saving series can be memory intensive, so if you don't need to save them you can uncheck the Save box.

See Also

Changing the Series Graph Type (see page 1787)

Changing Other Decision Graph Series Properties (see page 1786)

3.2.3.10.19 Setting Decision Graph Template Defaults

Decision graphs display the values from two dimensions of the decision cube: one dimension is displayed as an axis of the graph, and the other is used to create a set of series. The template for that dimension provides default properties for those series (such as whether the series are bar, line, area, and so on). As users pivot from one state to another, any required series for the dimension are created using the series type and other defaults specified in the template.

A separate template is provided for cases where users pivot to a state where only one dimension is active. A one-dimensional state is often represented with a pie chart, so a separate template is provided for this case.

You can

- Change the default graph type. (see page 1786)
- Change other graph template properties. (see page 1786)
- View and set overall graph properties. (see page 1794)

See Also

Customizing Decision Graph Series (see page 1799)

3.2.3.10.20 Setting Dimension State

The *ActiveFlag* property controls which dimensions get loaded. You can set this property on the Dimension Settings tab of the Decision Cube editor using the Activity Type control. When this control is set to *Active*, the dimension is loaded unconditionally, and will always take up space. Note that the number of dimensions in this state must always be less than *MaxDimensions*, and

the number of summaries set to *Active* must be less than *MaxSummaries*. You should set a dimension or summary to *Active* only when it is critical that it be available at all times. An *Active* setting decreases the ability of the cube to manage the available memory.

When *ActiveFlag* is set to *AsNeeded*, a dimension or summary is loaded only if it can be loaded without exceeding the *MaxDimensions*, *MaxSummaries*, or *MaxCells* limit. The decision cube will swap dimensions and summaries that are marked *AsNeeded* in and out of memory to keep within the limits imposed by *MaxCells*, *MaxDimensions*, and *MaxSummaries*. Thus, a dimension or summary may not be loaded in memory if it is not currently being used. Setting dimensions that are not used frequently to *AsNeeded* results in better loading and pivoting performance, although there will be a time delay to access dimensions which are not currently loaded.

See Also

Setting Maximum Dimensions (see page 1793)

Using Paged Dimensions (see page 1793)

3.2.3.10.21 Setting Maximum Dimensions, Summaries, and Cells

The decision cube's *MaxDimensions* and *MaxSummaries* properties can be used with the *CubeDim.ActiveFlag* property to control how many dimensions and summaries can be loaded at a time. You can set the maximum values on the Cube Capacity page of the Decision Cube editor to place some overall control on how many dimensions or summaries can be brought into memory at the same time.

Limiting the number of dimensions or summaries provides a rough limit on the amount of memory used by the decision cube. However, it does not distinguish between dimensions with many values and those with only a few. For greater control of the absolute memory demands of the decision cube, you can also limit the number of cells in the cube. Set the maximum number of cells on the Cube Capacity page of the Decision Cube editor.

See Also

Setting Dimension State (see page 1792)

Using Paged Dimensions (see page 1793)

3.2.3.10.22 Setting the Maximum Available Dimensions and Summaries

To determine the maximum number of dimensions and summaries available for decision pivots, decision grids, and decision graphs bound to the selected decision cube, display the Decision Cube editor and click the Memory Control tab. Use the edit controls to adjust the current settings, if necessary. These settings help to control the amount of memory required by the decision cube. For more information, see Decision Support Components and Memory Control. (see page 1800)

See Also

Viewing and Changing Dimension Settings (see page 1794)

Viewing and Changing Design Options (see page 1794)

3.2.3.10.23 Using Paged Dimensions

When Binning is set to Set on the Dimension Settings tab of the Decision cube editor and Start Value is not NULL, the dimension is said to be "paged," or "permanently drilled down." You can access data for just a single value of that dimension at a time, although you can programmatically access a series of values sequentially. Such a dimension may not be pivoted or opened.

It is extremely memory intensive to include dimensional data for dimensions that have very large numbers of values. By making such dimensions paged, you can display summary information for one value at a time. Information is usually easier to read when

displayed this way, and memory consumption is much easier to manage.

See Also

Setting Maximum Dimensions (see page 1793)

Setting Dimension State (see page 1792)

3.2.3.10.24 Viewing and Changing Design Options

To determine how much information appears at design time, display the Decision Cube editor and click the Memory Control tab. Then, check the setting that indicates which names and data to display. Display of data or field names at design time can cause performance delays in some cases because of the time needed to fetch the data.

See Also

Viewing and Changing Dimension Settings (see page 1794)

Setting the Maximum Available Dimensions and Summaries (see page 1793)

3.2.3.10.25 Viewing and Changing Dimension Settings

To view the dimension settings, display the Decision Cube editor and click the Dimension Settings tab. Then, select a dimension or summary in the Available Fields list. Its information appears in the boxes on the right side of the editor:

- To change the dimension or summary name that appears in the decision pivot, decision grid, or decision graph, enter a new name in the Display Name edit box.
- To determine whether the selected field is a dimension or summary, read the text in the Type edit box. If the dataset is a *TTable* component, you can use Type to specify whether the selected field is a dimension or summary.
- To disable or activate the selected dimension or summary, change the setting in the Active Type drop-down list box: Active, As Needed, or Inactive. Disabling a dimension or setting it to As Needed saves memory.
- To change the format of that dimension or summary, enter a format string in the Format edit box.
- To display that dimension or summary by Year, Quarter, or Month, change the setting in the Binning drop-down list box. Note that you can choose Set in the Binning list box to put the selected dimension or summary in a permanently "drilled down" state. This can be useful for saving memory when a dimension has many values. For more information, see Decision support components and memory control. (see page 1800)
- To determine the starting value for ranges, or the drill-down value for a "Set" dimension, first choose the appropriate Grouping value in the Grouping drop-down, and then enter the starting range value or permanent drill-down value in the Initial Value drop-down list.

See Also

Setting the Maximum Available Dimensions and Summaries (see page 1793)

Viewing and Changing Design Options (see page 1794)

3.2.3.10.26 Viewing Overall Decision Graph Properties

To view and set decision graph properties other than type and series

1. Select the Chart page at the top of the Chart Editing dialog box.
2. Choose the appropriate property tab and select settings.

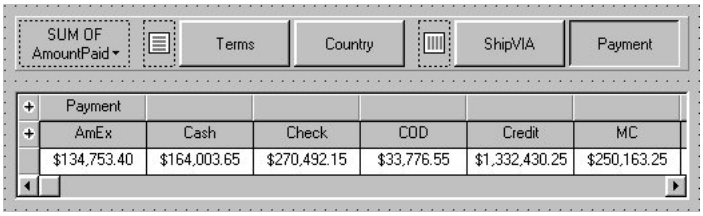
See Also

Changing the Default Decision Graph Type (see page 1786)

Changing Other Decision Graph Template Properties (see page 1786)

3.2.3.10.27 One-Dimensional Crosstabs

One-dimensional crosstabs show a summary row (or column) for the categories of a single dimension. For example, if Payment is the chosen column dimension and Amount Paid is the summary category, the crosstab in the following figure shows the amount paid using each method.



3.2.3.10.28 Using Decision Graphs

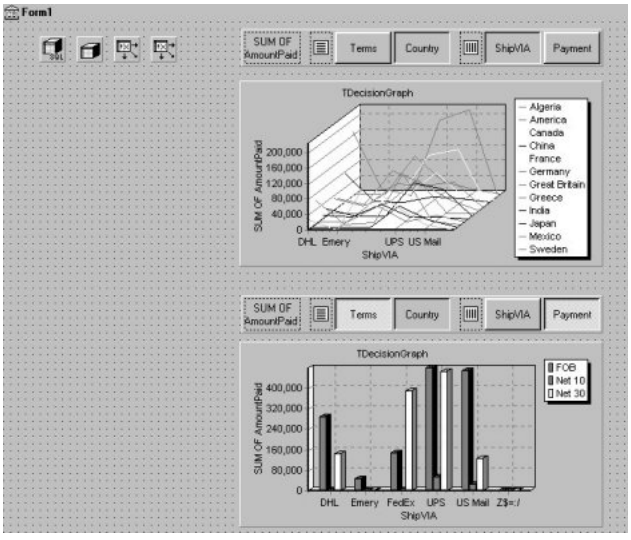
The decision graph component, TDecisionGraph, displays fields from the decision source TDecisionSource as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot TDecisionPivot.

Graphed data comes from a specially formatted dataset such as TDecisionQuery. For an overview of how the decision support components handle and arrange this data, see Using Decision Support Components. (see page 1800)

By default, the first row dimension appears as the x-axis and the first column dimension appears as the y-axis.

You can use decision graphs instead of or in addition to decision grids, which present cross-tabulated data in tabular form. Decision grids and decision graphs that are bound to the same decision source present the same data dimensions. To show different summary data for the same dimensions, you can bind more than one decision graph to the same decision source. To show different dimensions, bind decision graphs to different decision sources.

For example, in the following figure the first decision pivot and graph are bound to the first decision source and the second decision pivot and graph are bound to the second. So, each graph can show different dimensions.



For more information about what appears in a decision graph, see the next section, The Decision Graph Display. (see page 1796)

To create a decision graph, see the previous section, Creating Decision Graphs. (see page 1802)

For a discussion of decision graph properties and how to change the appearance and behavior of decision graphs, see Customizing Decision Graphs. (see page 1801)

See Also

Using Decision Grids (see page 1801)

3.2.3.10.29 The Decision Graph Display

By default, the decision graph plots summary values for categories in the first active row field (along the y-axis) against values in the first active column field (along the x-axis). Each graphed category appears as a separate series.

If only one dimension is selected—for example, by clicking only one *TDecisionPivot* button—only one series is graphed.

If you used a decision pivot, you can push its buttons to determine which decision cube fields (dimensions) are graphed. To exchange graph axes, drag the decision pivot dimension buttons from one side of the separator space to the other. If you have a one-dimensional graph with all buttons on one side of the separator space, you can use the Row or Column icon as a drop target for adding buttons to the other side of the separator and making the graph multidimensional.

If you only want one column and one row to be active at a time, you can set the *ControlType* property for *TDecisionSource* to *xtRadio*. Then, there can be only one active field at a time for each decision cube axis, and the decision pivot's functionality will correspond to the graph's behavior. *xtRadioEx* works the same as *xtRadio*, but does not allow the state where all row or all columns dimensions are closed.

When you have both a decision grid and graph connected to the same *TDecisionSource*, you'll probably want to set *ControlType* back to *xtCheck* to correspond to the more flexible behavior of *TDecisionGrid*.

See Also

Creating Decision Graphs (see page 1802)

Using Decision Graphs (see page 1795)

Customizing Decision Graphs (see page 1801)

3.2.3.10.30 Using the Decision Cube Editor

You can use the Decision Cube editor to set the DimensionMap properties of decision cubes. You can display the Decision Cube editor through the **Object Inspector**, as described in the previous section, or by right-clicking a decision cube on a form at design time and choosing Decision Cube editor.

The Decision Cube Editor dialog box has two tabs:

- Dimension Settings (see page 1794) used to activate or disable available dimensions, rename and reformat dimensions, put dimensions in a permanently drilled state, and set date ranges to display.
- Memory Control, used to set the maximum number of dimensions and summaries (see page 1793) that can be active at one time, to display information about memory usage, and to determine the names and data that appear at design time. (see page 1794)

See Also

Decision Cube Properties and Events (see page 1788)

3.2.3.10.31 Creating and Using Decision Graphs

Decision graph components, TDecisionGraph, present cross-tabulated data in graphic form. Each decision graph shows the

value of a single summary, such as Sum, Count, or Avg, charted for one or more dimensions. For more information on crosstabs, see One-dimensional crosstabs. (see page 1795) For illustrations of decision graphs at design time, see the figures Decision support components at design time (see page 1790) and Decision graphs bound to different decision sources. (see page 1795)

The following topics are discussed in this section:

- Creating Decision Graphs (see page 1802)
- Using Decision Graphs (see page 1795)
- The Decision Graph Display (see page 1796)
- Customizing Decision Graphs (see page 1801)

See Also

Using Datasets with Decision Support Components (see page 1799)

Using Decision Cubes (see page 1797)

Using Decision Sources (see page 1803)

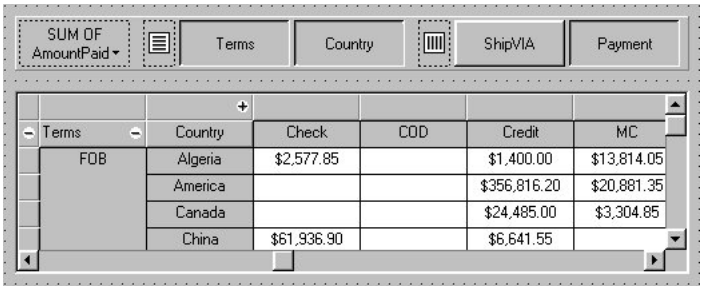
Using Decision Pivots (see page 1799)

Creating and Using Decision Grids (see page 1803)

3.2.3.10.32 Multidimensional Crosstabs

Multidimensional crosstabs use additional dimensions for the rows and/or columns. For example, a two-dimensional crosstab could show amounts paid by payment method for each country.

A three-dimensional crosstab could show amounts paid by payment method and terms by country, as shown in the following figure.



3.2.3.10.33 Using Decision Cubes

The decision cube component, TDecisionCube, is a multidimensional data store that fetches its data from a dataset (typically a specially structured SQL statement entered through *TDecisionQuery* or *TQuery*). The data is stored in a form that makes its easy to pivot (that is, change the way in which the data is organized and summarized) without needing to run the query a second time.

The following topics are discussed in this section:

- Decision Cube Properties and Events (see page 1788)
- Using the Decision Cube Editor (see page 1796)

See Also

Using Datasets with Decision Support Components (see page 1799)

Using Decision Sources (see page 1803)

Using Decision Pivots (see page 1799)

Creating and Using Decision Grids (see page 1803)

Creating and Using Decision Graphs (see page 1796)

3.2.3.10.34 Decision Grid Properties

The decision grid component, TDecisionGrid, displays data from the TDecisionSource component bound to TDecisionSource. By default, data appears in a grid with category fields on the left side and top of the grid.

The following are some special properties that control the appearance and behavior of decision grids:

- *TDecisionGrid* has unique properties for each dimension. To set these, choose *Dimensions* in the **Object Inspector**, then select a dimension. Its properties then appear in the **Object Inspector**: *Alignment* defines the alignment of category labels for that dimension, *Caption* can be used to override the default dimension name, *Color* defines the color of category labels, *FieldName* displays the name of the active dimension, *Format* can hold any standard format for that data type, and *Subtotals* indicates whether to display subtotals for that dimension. With summary fields, these same properties are used to changed the appearance of the data that appears in the summary area of the grid. When you're through setting dimension properties, either click a component in the form or choose a component in the drop-down list box at the top of the **Object Inspector**.
- The *Options* property of TDecisionGrid lets you control display of grid lines (*cgGridLines* = *True*), enabling of outline features (collapse and expansion of dimensions with + and - indicators; *cgOutliner* = *True*), and enabling of drag-and-drop pivoting (*cgPivotable* = *True*).
- The *OnDecisionDrawCell* event of *TDecisionGrid* gives you a chance to change the appearance of each cell as it is drawn. The event passes the *String*, *Font*, and *Color* of the current cell as reference parameters. You are free to alter those parameters to achieve effects such as special colors for negative values. In addition to the *DrawState* which is passed by *TCustomGrid*, the event passes *TDecisionDrawState*, which can be used to determine what type of cell is being drawn. Further information about the cell can be fetched using the *Cells*, *CellValueArray*, or *CellDrawState* functions.
- The *OnDecisionExamineCell* event of *TDecisionGrid* lets you hook the right-click-on-event to data cells, and is intended to allow a program to display information (such as detail records) about that particular data cell. When the user right-clicks a data cell, the event is supplied with all the information which is was used to compose the data value, including the currently active summary value and a *ValueArray* of all the dimension values which were used to create the summary value.

See Also

Creating Decision Grids (see page 1788)

Using Decision Grids (see page 1801)

3.2.3.10.35 Using Decision Pivots

The decision pivot component, TDecisionPivot, lets you open or close decision cube dimensions, or fields, by pressing buttons. When a row or column is opened by pressing a *TDecisionPivot* button, the corresponding dimension appears on the TDecisionGrid or TDecisionGraph component. When a dimension is closed, its detailed data doesn't appear; it collapses into the totals of other dimensions. A dimension may also be in a "drilled" state, where only the summaries for a particular value of the dimension field appear.

You can also use the decision pivot to reorganize dimensions displayed in the decision grid and decision graph. Just drag a button to the row or column area or reorder buttons within the same area.

For illustrations of decision pivots at design time, see the figures in Decision Support Components at Design Time (see page 1790)One-dimensional Crosstab (see page 1795) and Three-dimensional Crosstab. (see page 1797)

For information on special properties of TDecisionPivot, see Decision Pivot Properties. (see page 1791)

See Also

Using Datasets with Decision Support Components (see page 1799)

Using Decision Cubes (see page 1797)

Using Decision Sources (see page 1803)

Creating and Using Decision Grids (see page 1803)

Creating and Using Decision Graphs (see page 1796)

3.2.3.10.36 Customizing Decision Graph Series

The templates supply many defaults for each decision cube dimension, such as graph type and how series are displayed. Other defaults, such as series color, are defined by *TDecisionGraph*. If you want you can override the defaults for each series.

The templates are intended for use when you want the program to create the series for categories as they are needed, and discard them when they are no longer needed. If you want, you can set up custom series for specific category values. To do this, pivot the graph so its current display has a series for the category you want to customize. When the series is displayed on the graph, you can use the Chart editor to

- Change the Graph Type. (see page 1787)
- Change Other Series Properties. (see page 1786)
- Save Specific Graph Series that You Have Customized. (see page 1792)

To define series templates and set overall graph defaults, see Setting Decision Graph Template Defaults. (see page 1792)

3.2.3.10.37 Using Datasets with Decision Support Components

The only decision support component that binds directly to a dataset is the decision cube, TDecisionCube. *TDecisionCube* expects to receive data with groups and summaries defined by an SQL statement of an acceptable format. The GROUP BY phrase must contain the same non-summarized fields (and in the same order) as the SELECT phrase, and summary fields must be identified.

The decision query component, TDecisionQuery, is a specialized form of *TQuery*. You can use TDecisionQuery (see page 1802) to more simply define the setup of dimensions (rows and columns) and summary values used to supply data to decision cubes (*TDecisionCube*). The decision query has no properties than are not inherited from other components. Important inherited properties are Active and SQL.

You can also use a TQuery or TTable component (see page 1787) as a dataset for *TDecisionCube*, but the correct setup of the dataset and TDecisionCube are then the responsibility of the designer.

To work correctly with the decision cube, all projected fields in the dataset must either be dimensions or summaries. The summaries should be additive values (like sum or count), and should represent totals for each combination of dimension values. For maximum ease of setup, sums should be named "Sum..." in the dataset while counts should be named "Count...".

The Decision Cube can pivot, subtotal, and drill-in correctly only for summaries whose cells are additive. (SUM and COUNT are additive, while AVERAGE, MAX, and MIN are not.) Build pivoting crosstab displays only for grids that contain only additive aggregators. If you are using non-additive aggregators, use a static decision grid that does not pivot, drill, or subtotal.

Since averages can be calculated using SUM divided by COUNT, a pivoting average is added automatically when SUM and COUNT dimensions for a field are included in the dataset. Use this type of average in preference to an average calculated using an AVERAGE statement.

Averages can also be calculated using COUNT(*). To use COUNT(*) to calculate averages, include a "COUNT(*) COUNTALL" selector in the query. If you use COUNT(*) to calculate averages, the single aggregator can be used for all fields. Use COUNT(*) only in cases where none of the fields being summarized include blank values, or where a COUNT aggregator is not available for every field.

See Also

Using Decision Cubes (see page 1797)

Using Decision Sources (see page 1803)

Using Decision Pivots

Creating and Using Decision Grids (see page 1803)

Creating and Using Decision Graphs (see page 1796)

Working with Queries (see page 1674)

3.2.3.10.38 Using Decision Support Components

The decision support components help you create cross-tabulated—or, crosstab—tables and graphs. You can then use these tables and graphs to view and summarize data from different perspectives. For more information on cross-tabulated data, see About crosstabs. (see page 1804)

The following topics are discussed in this section:

- Overview of Decision Support Components (see page 1790)
- Guidelines for Using Decision Support Components (see page 1804)
- Decision Support Components at Runtime (see page 1789)
- Decision Support Components and Memory Control (see page 1800)

See Also

Designing Database Applications (see page 1566)

Using Data Controls (see page 1778)

Working with Field Components (see page 1877)

Understanding Datasets (see page 1632)

Connecting to Databases (see page 1506)

Using the Borland Database Engine (see page 1643)

3.2.3.10.39 Decision Support Components and Memory Control

When a dimension or summary is loaded into the decision cube, it takes up memory. Adding a new summary increases memory consumption linearly: that is, a decision cube with two summaries uses twice as much memory as the same cube with only one summary, a decision cube with three summaries uses three times as much memory as the same cube with one summary, and so on. Memory consumption for dimensions increases more quickly. Adding a dimension with 10 values increases memory consumption by a factor of 10. Adding a dimension with 100 values increases memory consumption 100 times. Thus adding dimensions to a decision cube can have a dramatic effect on memory use, and can quickly lead to performance problems. This effect is especially pronounced when adding dimensions that have many values.

Memory consumption can be limited by the following techniques:

- Setting maximum Dimensions (see page 1793)
- Setting Dimension State (see page 1792)
- Using Paged Dimensions (see page 1793)

The decision support components have a number of settings to help you control how and when memory is used. For more

information on the properties and techniques mentioned here, see TDecisionCube.

See Also

Overview of Decision Support Components (see page 1790)

About Crosstabs (see page 1804)

Guidelines for Using Decision Support Components (see page 1804)

Decision Support Components at Runtime (see page 1789)

3.2.3.10.40 Using Decision Grids

The decision grid component, TDecisionGrid, displays data from decision cubes TDecisionCube bound to decision sources TDecisionSource.

By default, the grid appears with dimension fields at its left side and/or top, depending on the grouping instructions defined in the dataset. Categories, one for each data value, appear under each field. You can

- Open and Close Dimensions (see page 1790)
- Reorganize (see page 1792)
- Drill Down for Detail (see page 1790)
- Limit Dimension Selection to a Single Dimension for Each Axis (see page 1790)

For more information about special properties and events of the decision grid, see Decision grid properties. (see page 1798)

See Also

Creating Decision Grids (see page 1788)

3.2.3.10.41 Customizing Decision Graphs

The decision graph component, TDecisionGraph, displays fields from the decision source (TDecisionSource) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot TDecisionPivot. You can change the type, colors, marker types for line graphs, and many other properties of decision graphs.

To customize a graph

1. Right-click it and choose Edit Chart. The Chart Editing dialog box appears.
2. Use the Chart page of the Chart Editing dialog box to view a list of visible series, select the series definition to use when two or more are available for the same series, change graph types for a template or series, and set overall graph properties. The Series list on the Chart page shows all decision cube dimensions (preceded by Template:) and currently visible categories. Each category, or series, is a separate object. You can:

- Add or delete series derived from existing decision-graph series. Derived series can provide annotations for existing series or represent values calculated from other series.
- Change the default graph type, and change the title of templates and series.

3. Use the Series page to establish dimension templates, then customize properties for each individual graph series.

By default, all series are graphed as bar graphs and up to 16 default colors are assigned. You can edit the template type and properties to create a new default. Then, as you pivot the decision source to different states, the template is used to dynamically create the series for each new state. For template details, see Setting decision graph template defaults. (see page 1792).

To customize individual series, follow the instructions in Customizing decision graph series. (see page 1799)

See Also

Creating Decision Graphs (see page 1802)

Using Decision Graphs (see page 1795)

The Decision Graph Display (see page 1796)

3.2.3.10.42 Creating Decision Graphs

To create a form with one or more decision graphs

1. Follow steps 1–3 listed under Guidelines for using decision support components. (see page 1804)
2. Add one or more decision graph components TDecisionGraph and bind them to the decision source, TDecisionSource, with the **Object Inspector** by setting their *DecisionSource* property to the appropriate decision source component.
3. Continue with steps 5–7 listed under Guidelines for using decision support components. (see page 1804)
4. Finally, right-click the graph and choose Edit Chart to modify the appearance of the graph series. You can set template properties for each graph dimension, then set individual series properties to override these defaults. For details, see Customizing decision graphs. (see page 1801)

For a description of what appears in the decision graph and how to use it, see Using decision graphs. (see page 1795)

To add a decision grid—or crosstab table—to the form, follow the instructions in Creating and using decision grids. (see page 1803)

See Also

The Decision Graph Display (see page 1796)

3.2.3.10.43 Creating Decision Datasets with the Decision Query Editor

All data used by the decision support components passes through the decision cube, which accepts a specially formatted dataset most easily produced by an SQL query. See Using datasets with decision support components (see page 1799) for more information.

While both *TTable* and *TQuery* can be used as decision datasets, it is easier to use TDecisionQuery; the Decision Query editor supplied with it can be used to specify tables, fields, and summaries to appear in the decision cube and will help you set up the SELECT and GROUP BY portions of the SQL correctly.

To use the Decision Query editor

1. Select the decision query component on the form, then right-click and choose Decision Query editor. The Decision Query editor dialog box appears.
2. Choose the database to use.
3. For single-table queries, click the Select Table button. For more complex queries involving multi-table joins, click the Query Builder button to display the SQL Builder or type the SQL statement into the edit box on the SQL tab page.
4. Return to the Decision Query editor dialog box.
5. In the Decision Query editor dialog box, select fields in the Available Fields list box and assign them to be either Dimensions or Summaries by clicking the appropriate right arrow button. As you add fields to the Summaries list, select from the menu displayed the type of summary to use: sum, count, or average.
6. By default, all fields and summaries defined in the SQL property of the decision query appear in the Active Dimensions and Active Summaries list boxes. To remove a dimension or summary, select it in the list and click the left arrow beside the list, or double-click the item to remove. To add it back, select it in the Available Fields list box and click the appropriate right arrow.

Once you define the contents of the decision cube, you can further manipulate dimension display with its *DimensionMap* property and the buttons of *TDecisionPivot*. For more information, see Using decision cubes (see page 1797),Using decision sources (see page 1803), and Using decision pivots (see page 1798).

Note: When you use the Decision Query editor, the query is initially handled in ANSI-92 SQL syntax, then translated (if necessary) into the dialect used by the server. The Decision Query editor reads and displays only ANSI standard SQL. The

dialect translation is automatically assigned to the *TDecisionQuery*'s SQL property. To modify a query, edit the ANSI-92 version in the Decision Query rather then the SQL property.

See Also

Creating Decision Datasets with TQuery or TTable (see page 1787)

3.2.3.10.44 Using Decision Sources

The decision source component, TDecisionSource, defines the current pivot state of decision grids or decision graphs. Any two objects which use the same decision source also share pivot states.

The following are some special properties and events that control the appearance and behavior of decision sources:

- The ControlType property of *TDecisionSource* indicates whether the decision pivot buttons should act like check boxes (multiple selections) or radio buttons (mutually exclusive selections).
- The SparseCols and SparseRows properties of *TDecisionSource* indicate whether to display columns or rows with no values; if *True*, sparse columns or rows are displayed.

TDecisionSource has the following events:

- OnLayoutChange occurs when the user performs pivots or drill-downs that reorganize the data.
- OnNewDimensions occurs when the data is completely altered, such as when the summary or dimension fields are altered.
- OnSummaryChange occurs when the current summary is changed.
- OnStateChange occurs when the Decision Cube activates or deactivates.
- OnBeforePivot occurs when changes are committed but not yet reflected in the user interface. Developers have an opportunity to make changes, for example, in capacity or pivot state, before application users see the result of their previous action.
- OnAfterPivot fires after a change in pivot state. Developers can capture information at that time.

See Also

Using Datasets with Decision Support Components (see page 1799)

Using Decision Cubes (see page 1797)

Using Decision Pivots (see page 1799)

Creating and Using Decision Grids (see page 1803)

Creating and Using Decision Graphs (see page 1796)

3.2.3.10.45 Creating and Using Decision Grids

Decision grid components, TDecisionGrid, present cross-tabulated data in table form. These tables are also called crosstabs, described in About crosstabs. (see page 1804) The figure Decision support components at design time (see page 1790) shows a decision grid on a form at design time.

The following topics are discussed in this section:

- Creating Decision Grids (see page 1788)
- Using Decision Grids (see page 1801)
- Decision Grid Properties (see page 1798)

See Also

Using Datasets with Decision Support Components (see page 1799)

Using Decision Cubes (see page 1797)

Using Decision Sources (see page 1803)

Using Decision Pivots (see page 1799)

Creating and Using Decision Graphs (see page 1796)

3.2.3.10.46 About Crosstabs

Cross-tabulations, or crosstabs, are a way of presenting subsets of data so that relationships and trends are more visible. Table fields become the dimensions of the crosstab while field values define categories and summaries within a dimension.

You can use the decision support components to set up crosstabs in forms. TDecisionGrid shows data in a table, while TDecisionGraph charts it graphically. TDecisionPivot has buttons that make it easier to display and hide dimensions and move them between columns and rows.

Crosstabs can be one-dimensional or multidimensional.

The following topics are discussed in this section:

- One-Dimensional Crosstabs (see page 1795)
- Multidimensional Crosstabs (see page 1797)

See Also

Overview of Decision Support Components (see page 1790)

Guidelines for Using Decision Support Components (see page 1804)

Decision Support Components at Runtime (see page 1789)

Decision Support Components and Memory Control (see page 1800)

3.2.3.10.47 Guidelines for Using Decision Support Components

The decision support components listed in Overview of decision support components (see page 1790) can be used together to present multidimensional data as tables and graphs. More than one grid or graph can be attached to each dataset. More than one instance of *TDecisionPivot* can be used to display the data from different perspectives at runtime.

To create a form with tables and graphs of multidimensional data

1. Create a form.
2. Add these components to the form and use the **Object Inspector** to bind them as indicated:

- A dataset, usually *TDecisionQuery* (for details, see Creating Decision Datasets with The Decision Query Editor (see page 1802)) or TQuery
- A decision cube, *TDecisionCube*, bound to the dataset by setting its DataSet property to the dataset's name
- A decision source, *TDecisionSource*, bound to the decision cube by setting its *DecisionCube* property to the decision cube's name

3. Add a decision pivot, *TDecisionPivot*, and bind it to the decision source with the **Object Inspector** by setting its *DecisionSource* property to the appropriate decision source name. The decision pivot is optional but useful; it lets the form developer and end users change the dimensions displayed in decision grids or decision graphs by pushing buttons. In its default orientation, horizontal, buttons on the left side of the decision pivot apply to fields on the left side of the decision grid (rows); buttons on the right side apply to fields at the top of the decision grid (columns). You can determine where the decision pivot's buttons appear by setting its GroupLayout property to *xtVertical*, *xtLeftTop*, or *xtHorizontal* (the default). For more information on decision pivot properties, see Using decision pivots. (see page 1798)
4. Add one or more decision grids and graphs, bound to the decision source. For details, see Creating and using decision grids (see page 1803) and Creating and using decision graphs (see page 1796).

5. Use the Decision Query editor or SQL property of *TDecisionQuery* (or *TQuery*) to specify the tables, fields, and summaries to display in the grid or graph. The last field of the SQL SELECT should be the summary field. The other fields in the SELECT must be GROUP BY fields. For instructions, see Creating decision datasets with the Decision Query editor (see page 1802).
6. Set the *Active* property of the decision query (or alternate dataset component) to *True*.
7. Use the decision grid and graph to show and chart different data dimensions. See Using decision grids (see page 1803) and Using decision graphs. (see page 1796) for instructions and suggestions

For an illustration of all decision support components on a form, see the figure Decision support components at design time. (see page 1790)

See Also

About Crosstabs (see page 1804)

Decision Support Components at Runtime (see page 1789)

Decision Support Components and Memory Control (see page 1800)

3.2.3.11 Using provider components

Topics

Name	Description
Responding to Client Update Requests (see page 1808)	A provider applies updates to database records based on a *Delta* data packet received from a client dataset or XML broker. The client requests updates by calling the ApplyUpdates method (indirectly, through the *IAppServer* interface). As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information (see page 1540) with a client dataset before and after the call to *ApplyUpdates*. This communication takes place using the BeforeApplyUpdates and AfterApplyUpdates event handlers. If you are using a dataset provider, a number of additional events allow you more control: When a dataset provider receives an update request, it... more (see page 1808)
Adding Custom Information to Data Packets (see page 1808)	Dataset providers can add application-defined information to data packets using the OnGetDataSetProperties event. This information is encoded as an OleVariant, and stored under a name you specify. Client datasets can then retrieve the information using their GetOptionalParam method. You can also specify that the information be included in delta packets that the client dataset sends when updating records. In this case, the client dataset may never be aware of the information, but the provider can send a round-trip message to itself. When adding custom information in the *OnGetDataSetProperties* event, each individual attribute (sometimes called an "optional parameter") is specified using... more (see page 1808)
Applying Updates to Datasets That do Not Represent a Single Table (see page 1809)	When a dataset provider generates SQL statements that apply updates directly to a database server, it needs the name of the database table that contains the records. This can be handled automatically for many datasets such as table type datasets or "live" *TQuery* components. Automatic updates are a problem however, if the provider must apply updates to the data underlying a stored procedure with a result set or a multi-table query. There is no easy way to obtain the name of the table to which updates should be applied. If the query or stored procedure is a BDE-enabled dataset (... more (see page 1809)
Choosing How to Apply Updates Using a Dataset Provider (see page 1810)	*TXMLTransformProvider* components always apply updates to the associated XML document. When using *TDataSetProvider*, however, you can choose how updates are applied. By default, when *TDataSetProvider* components apply updates and resolve update errors, they communicate directly with the database server using dynamically generated SQL statements. This approach has the advantage that your server application does not need to merge updates twice (first to the dataset, and then to the remote server). However, you may not always want to take this approach. For example, you may want to use some of the events on the dataset component. Alternately, the dataset you... more (see page 1810)

Communicating with the Client Dataset (see page 1811)	All communication between a provider and a client dataset or XML broker takes place through an IAppServer interface. If the provider is in the same application as the client, this interface is implemented by a hidden object generated automatically for you, or by a TLocalConnection component. If the provider is part of a multi-tiered application, this is the interface for the application server's remote data module or (in the case of a SOAP server) an interface generated by the connection component. Most applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client... more (see page 1811)
Controlling What Information Is Included in Data Packets (see page 1812)	When working with a dataset provider, there are a number of ways to control what information is included in data packets that are sent to and from the client. These include • Specifying what fields appear in data packets (see page 1819) • Setting options that influence the data packets (see page 1818) • Adding custom information to data packets (see page 1808) **Note:** These techniques for controlling the content of data packets are only available for dataset providers. When using *TXMLTransformProvider*, you can only control the content of data packets by controlling the transformation file the provider uses.
Determining the Source of Data (see page 1812)	When you use a provider component, you must specify the source it uses to get the data it assembles into data packets. Depending on your version of Delphi, you can specify the source as one of the following: • To provide the data from a dataset, use TDataSetProvider. • To provide the data from an XML document, use TXMLTransformProvider.
Editing Delta Packets Before Updating the Database (see page 1813)	Before a dataset provider applies updates to the database, it generates an OnUpdateData event. The *OnUpdateData* event handler receives a copy of the *Delta* packet as a parameter. This is a client dataset. In the *OnUpdateData* event handler, you can use any of the properties and methods of the client dataset to edit the *Delta* packet before it is written to the dataset. One particularly useful property is the UpdateStatus property. *UpdateStatus* indicates what type of modification the current record in the delta packet represents. It can have any of the values in the following table: ***UpdateStatus values***
Handling Server Constraints (see page 1814)	Most relational database management systems implement constraints on their tables to enforce data integrity. A constraint is a rule that governs data values in tables and columns, or that governs data relationships across columns in different tables. For example, most SQL-92 compliant relational databases support the following constraints: • NOT NULL, to guarantee that a value supplied to a column has a value. • NOT NULL UNIQUE, to guarantee that column value has a value and does not duplicate any other value already in that column for another record. • CHECK, to guarantee that a value supplied to a column falls within a... more (see page 1814)
Influencing How Updates Are Applied (see page 1815)	The OnUpdateData event gives your dataset provider a chance to indicate how records in the delta packet are applied to the database. By default, changes in the delta packet are written to the database using automatically generated SQL UPDATE, INSERT, or DELETE statements such as

Resolving Update Errors On the Provider (see page 1816)	When an error condition arises as the dataset provider tries to post a record in the delta packet, an OnUpdateError event occurs. If the provider can't resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered, and copies the unresolved records into a results data packet that it passes back to the client for further reconciliation. In multi-tiered applications, this mechanism lets you handle any update errors you can resolve mechanically on the application server, while still allowing user interaction on the... more (see page 1816)
Responding to Client Data Requests (see page 1816)	Usually client requests for data are handled automatically. A client dataset or XML broker requests a data packet by calling GetRecords (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the associated dataset or XML document, creating a data packet, and sending the packet to the client. The provider has the option of editing data after it has been assembled into a data packet but before the packet is sent to the client. For example, you might want to remove records from the packet based on some criterion (such as the user's level of access), or,... more (see page 1816)
Responding to Client-generated Events (see page 1817)	Provider components implement a general-purpose event that lets you create your own calls from client datasets directly to the provider. This is the OnDataRequest event. *OnDataRequest* is not part of the normal functioning of the provider. It is simply a hook to allow your client datasets to communicate directly with providers. The event handler takes an OleVariant as an input parameter and returns an OleVariant. By using OleVariants, the interface is sufficiently general to accommodate almost any information you want to pass to or from the provider. To generate an *OnDataRequest* event, the client application calls the DataRequest method... more (see page 1817)
Screening Individual Updates (see page 1817)	Immediately before each update is applied, a dataset provider receives a BeforeUpdateRecord event. You can use this event to edit records before they are applied, similar to the way you can use the *OnUpdateData* event to edit entire delta packets (see page 1813). For example, the provider does not compare BLOB fields (such as memos) when checking for update conflicts. If you want to check for update errors involving BLOB fields, you can use the *BeforeUpdateRecord* event. In addition, you can use this event to apply updates yourself or to screen and reject updates. The *BeforeUpdateRecord* event handler lets you signal that... more (see page 1817)
Setting Options That Influence the Data Packets (see page 1818)	The Options property of a dataset provider lets you specify when BLOBs or nested detail tables are sent, whether field display properties are included, what type of updates are allowed, and so on. The following table lists the possible values that can be included in *Options*. ***Provider options***
Specifying What Fields Appear in Data Packets (see page 1819)	When using a dataset provider, you can control what fields are included in data packets by creating persistent fields (see page 1859) on the dataset that the provider uses to build data packets. The provider then includes only these fields. Fields whose values are generated dynamically by the source dataset (such as calculated fields or lookup fields) can be included, but appear to client datasets on the receiving end as static read-only fields. If the client dataset will be editing the data and applying updates, you must include enough fields so that there are no duplicate records in the data packet. Otherwise, when... more (see page 1819)
Using Provider Components (see page 1819)	Provider components (TDataSetProvider and TXMLTransformProvider) supply the most common mechanism by which client datasets obtain their data. Providers • Receive data requests from a client dataset (or XML broker), fetch the requested data, package the data into a transportable data packet, and return the data to the client dataset (or XML broker). This activity is called "providing." • Receive updated data from a client dataset (or XML broker), apply updates to the database server, source dataset, or source XML document, and log any updates that cannot be applied, returning unresolved updates to the client dataset for further reconciliation. This activity is... more (see page 1819)

3.2.3.11.1 Responding to Client Update Requests

A provider applies updates to database records based on a *Delta* data packet received from a client dataset or XML broker. The client requests updates by calling the ApplyUpdates method (indirectly, through the *IAppServer* interface).

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information (see page 1540) with a client dataset before and after the call to *ApplyUpdates*. This communication takes place using the BeforeApplyUpdates and AfterApplyUpdates event handlers.

If you are using a dataset provider, a number of additional events allow you more control:

When a dataset provider receives an update request, it generates an *OnUpdateData* event, where you can edit the Delta packet (see page 1813) before it is written to the dataset or influence how updates are applied (see page 1815). After the *OnUpdate*Data event, the provider writes the changes to the database or source dataset.

The provider performs the update on a record-by-record basis. Before the dataset provider applies each record, it generates a *BeforeUpdateRecord* event, which you can use to screen updates (see page 1817) before they are applied. If an error occurs when updating a record, the provider receives an *OnUpdateError* event where it can resolve the error (see page 1816). Usually errors occur because the change violates a server constraint or a database record was changed by a different application subsequent to its retrieval by the provider, but prior to the client dataset's request to apply updates.

Update errors can be processed by either the dataset provider or the client dataset. When the provider is part of a multi-tiered application, it should handle all update errors that do not require user interaction to resolve. When the provider can't resolve an error condition, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it returns to the client dataset for further reconciliation.

The event handlers for all provider events are passed the set of updates as a client dataset. If your event handler is only dealing with certain types of updates, you can filter the dataset based on the update status of records. By filtering the records, your event handler does not need to sort through records it won't be using. To filter the client dataset on the update status of its records, set its StatusFilter property.

Note: Applications must supply extra support when the updates are directed at a dataset that does not represent a single table (see page 1809).

See Also

Controlling What Information Is Included in Data Packets (see page 1812)

Responding to Client-generated Events (see page 1817)

Handling Server Constraints (see page 1814)

Responding to Client Data Requests (see page 1816)

Choosing How to Apply Updates Using a Dataset Provider (see page 1810)

Determining the Source of Data (see page 1812)

Communicating with the Client Dataset (see page 1811)

3.2.3.11.2 Adding Custom Information to Data Packets

Dataset providers can add application-defined information to data packets using the OnGetDataSetProperties event. This information is encoded as an OleVariant, and stored under a name you specify. Client datasets can then retrieve the information using their GetOptionalParam method. You can also specify that the information be included in delta packets that the client dataset sends when updating records. In this case, the client dataset may never be aware of the information, but the provider

can send a round-trip message to itself.

When adding custom information in the *OnGetDataSetProperties* event, each individual attribute (sometimes called an "optional parameter") is specified using a Variant array that contains three elements: the name (a string), the value (a Variant), and a boolean flag indicating whether the information should be included in delta packets when the client applies updates. Add multiple attributes by creating a Variant array of Variant arrays. For example, the following *OnGetDataSetProperties* event handler sends two values, the time the data was provided and the total number of records in the source dataset. Only the time the data was provided is returned when client datasets apply updates:

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject; DataSet: TDataSet; out
Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
void __fastcall TMyDataModule1::Provider1GetDataSetProperties(TObject *Sender, TDataSet
*DataSet, out OleVariant Properties)
{
  int ArrayBounds[2];
  ArrayBounds[0] = 0;
  ArrayBounds[1] = 1;
  Properties = VarArrayCreate(ArrayBounds, 1, varVariant);
  Variant values[3];
  values[0] = Variant("TimeProvided");
  values[1] = Variant(Now());
  values[2] = Variant(true);
  Properties[0] = VarArrayOf(values,2);
  values[0] = Variant("TableSize");
  values[1] = Variant(DataSet->RecordCount);
  values[2] = Variant(false);
  Properties[1] = VarArrayOf(values,2);
}
```

When the client dataset applies updates, the time the original records were provided can be read in the provider's OnUpdateData event:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet: TCustomClientDataSet);
var
  WhenProvided: TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender, TCustomClientDataSet
*DataSet)
{
  Variant WhenProvided = DataSet->GetOptionalParam("TimeProvided");
  ...
}
```

See Also

Specifying What Fields Appear in Data Packets (see page 1819)

Setting Options That Influence the Data Packets (see page 1818)

Adding Application Specific Information to the Data (see page 1709)

3.2.3.11.3 Applying Updates to Datasets That do Not Represent a Single Table

When a dataset provider generates SQL statements that apply updates directly to a database server, it needs the name of the database table that contains the records. This can be handled automatically for many datasets such as table type datasets or

"live" *TQuery* components. Automatic updates are a problem however, if the provider must apply updates to the data underlying a stored procedure with a result set or a multi-table query. There is no easy way to obtain the name of the table to which updates should be applied.

If the query or stored procedure is a BDE-enabled dataset (*TQuery* or *TStoredProc*) and it has an associated update object, the provider uses the update object. However, if there is no update object, you can supply the table name programmatically in an OnGetTableName event handler. Once an event handler supplies the table name, the provider can generate appropriate SQL statements to apply updates.

Supplying a table name only works if the target of the updates is a single database table (that is, only the records in one table need to be updated). If the update requires making changes to multiple underlying database tables, you must explicitly apply the updates in code using the BeforeUpdateRecord event of the provider. Once this event handler has applied an update, you can set the event handler's *Applied* parameter to *True* so that the provider does not generate an error.

Note: If the provider is associated with a BDE-enabled dataset, you can use an update object (see page 1692) in the *BeforeUpdateRecord* event handler to apply updates using customized SQL statements.

See Also

Resolving Update Errors On the Provider (see page 1816)

Editing Delta Packets Before Updating the Database (see page 1813)

Influencing How Updates Are Applied (see page 1815)

Screening Individual Updates (see page 1817)

3.2.3.11.4 Choosing How to Apply Updates Using a Dataset Provider

TXMLTransformProvider components always apply updates to the associated XML document. When using *TDataSetProvider*, however, you can choose how updates are applied. By default, when *TDataSetProvider* components apply updates and resolve update errors, they communicate directly with the database server using dynamically generated SQL statements. This approach has the advantage that your server application does not need to merge updates twice (first to the dataset, and then to the remote server).

However, you may not always want to take this approach. For example, you may want to use some of the events on the dataset component. Alternately, the dataset you use may not support the use of SQL statements (for example if you are providing from a *TClientDataSet* component.

TDataSetProvider lets you decide whether to apply updates to the database server using SQL or to the source dataset by setting the ResolveToDataSet property. When this property is *True*, updates are applied to the dataset. When it is *False*, updates are applied directly to the underlying database server.

See Also

Responding to Client Data Requests (see page 1816)

Responding to Client-generated Events (see page 1817)

Handling Server Constraints (see page 1814)

Responding to Client Update Requests (see page 1808)

Controlling What Information Is Included in Data Packets (see page 1812)

Determining the Source of Data (see page 1812)

Communicating with the Client Dataset (see page 1811)

3.2.3.11.5 Communicating with the Client Dataset

All communication between a provider and a client dataset or XML broker takes place through an IAppServer interface. If the provider is in the same application as the client, this interface is implemented by a hidden object generated automatically for you, or by a TLocalConnection component. If the provider is part of a multi-tiered application, this is the interface for the application server's remote data module or (in the case of a SOAP server) an interface generated by the connection component.

Most applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client dataset or XML broker. However, when necessary, you can make direct calls to the *IAppServer* interface by using the AppServer property of a client dataset.

The following table lists the methods of the *IAppServer* interface, as well as the corresponding methods and events on the provider component and the client dataset. These *IAppServer* methods include a *Provider* parameter. In multi-tiered applications, this parameter indicates the provider on the application server with which the client dataset communicates. Most methods also include an OleVariant parameter called *OwnerData* that allows a client dataset and a provider to pass custom information back and forth. *OwnerData* is not used by default, but is passed to all event handlers so that you can write code that allows your provider to adjust to application-defined information before and after each call from a client dataset.

AppServer interface members

IAppServer	Provider Component	TClientDataSet
AS_ApplyUpdatesAS_ApplyUpdates method	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event.
AS_DataRequest method	DataRequest method, OnDataRequest event	DataRequest method.
AS_Execute method	Execute method, BeforeExecute event, AfterExecute event	Execute method, BeforeExecute event, AfterExecute event.
AS_GetParams method	GetParams method, BeforeGetParams event, AfterGetParams event	FetchParams method, BeforeGetParams event, AfterGetParams event.
AS_GetProviderNames method	Used to identify all available providers.	Used to create a design-time list for ProviderName property.
AS_GetRecords method	GetRecords method, BeforeGetRecords event, AfterGetRecords event	GetNextPacket method, Data property, BeforeGetRecords event, AfterGetRecords event
AS_RowRequest method	RowRequest method, BeforeRowRequest event, AfterRowRequest event	FetchBlobs method, FetchDetails method, RefreshRecord method, BeforeRowRequest event, AfterRowRequest event

See Also

Responding to Client Data Requests (see page 1816)

Responding to Client-generated Events (see page 1817)

Handling Server Constraints (see page 1814)

Responding to Client Update Requests (see page 1808)

Controlling What Information Is Included in Data Packets (see page 1812)

Choosing How to Apply Updates Using a Dataset Provider (see page 1810)

Determining the Source of Data (see page 1812)

3.2.3.11.6 Controlling What Information Is Included in Data Packets

When working with a dataset provider, there are a number of ways to control what information is included in data packets that are sent to and from the client. These include

- Specifying what fields appear in data packets (see page 1819)
- Setting options that influence the data packets (see page 1818)
- Adding custom information to data packets (see page 1808)

Note: These techniques for controlling the content of data packets are only available for dataset providers. When using *TXMLTransformProvider*, you can only control the content of data packets by controlling the transformation file the provider uses.

See Also

Responding to Client Data Requests (see page 1816)

Responding to Client-generated Events (see page 1817)

Handling Server Constraints (see page 1814)

Responding to Client Update Requests (see page 1808)

Choosing How to Apply Updates Using a Dataset Provider (see page 1810)

Determining the Source of Data (see page 1812)

Communicating with the Client Dataset (see page 1811)

3.2.3.11.7 Determining the Source of Data

When you use a provider component, you must specify the source it uses to get the data it assembles into data packets. Depending on your version of Delphi, you can specify the source as one of the following:

- To provide the data from a dataset, use TDataSetProvider.
- To provide the data from an XML document, use TXMLTransformProvider.

Using a dataset as the source of the data

If the provider is a dataset provider (*TDataSetProvider*), set the DataSet property of the provider to indicate the source dataset. At design time, select from available datasets in the *DataSet* property drop-down list in the **Object Inspector**.

TDataSetProvider interacts with the source dataset using the IProviderSupport interface. This interface is introduced by *TDataSet*, so it is available for all datasets. However, the *IProviderSupport* methods implemented in *TDataSet* are mostly stubs that don't do anything or that raise exceptions.

The dataset classes that ship with Delphi (BDE-enabled datasets, ADO-enabled datasets, *dbExpress* datasets, and InterBase Express datasets) override these methods to implement the *IProviderSupport* interface in a more useful fashion. Client datasets don't add anything to the inherited *IProviderSupport* implementation, but can still be used as a source dataset as long as the *ResolveToDataSet* property of the provider is *True*.

Component writers that create their own custom descendants from *TDataSet* must override all appropriate *IProviderSupport* methods if their datasets are to supply data to a provider. If the provider only provides data packets on a read-only basis (that is, if it does not apply updates), the *IProviderSupport* methods implemented in *TDataSet* may be sufficient.

Using an XML document as the source of the data

If the provider is an XML provider, set the XMLDataFile property of the provider to indicate the source document.

XML providers must transform the source document into data packets, so in addition to indicating the source document, you

must also specify how to transform that document into data packets. This transformation is handled by the provider's TransformRead property. *TransformRead* represents a *TXMLTransform* object. You can set its properties to specify what transformation to use, and use its events to provide your own input to the transformation. For more information on using XML providers, see Using an XML document as the source for a provider (see page 1846).

See Also

Responding to Client Data Requests (see page 1816)

Responding to Client-generated Events (see page 1817)

Handling Server Constraints (see page 1814)

Responding to Client Update Requests (see page 1808)

Controlling What Information Is Included in Data Packets (see page 1812)

Choosing How to Apply Updates Using a Dataset Provider (see page 1810)

Communicating with the Client Dataset (see page 1811)

3.2.3.11.8 Editing Delta Packets Before Updating the Database

Before a dataset provider applies updates to the database, it generates an OnUpdateData event. The *OnUpdateData* event handler receives a copy of the *Delta* packet as a parameter. This is a client dataset.

In the *OnUpdateData* event handler, you can use any of the properties and methods of the client dataset to edit the *Delta* packet before it is written to the dataset. One particularly useful property is the UpdateStatus property. *UpdateStatus* indicates what type of modification the current record in the delta packet represents. It can have any of the values in the following table:

UpdateStatus values

Value	Description
usUnmodified	Record contents have not been changed
usModified	Record contents have been changed
usInserted	Record has been inserted
usDeleted	Record has been deleted

For example, the following *OnUpdateData* event handler inserts the current date into every new record that is inserted into the database:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet: TCustomClientDataSet);
begin
  with DataSet do
  begin
    First;
    while not Eof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
        FieldByName('DateCreated').AsDateTime := Date;
        Post;
      end;
      Next;
    end;
end;
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender, TCustomClientDataSet
*DataSet)
{
```

```
    DataSet->First();
    while (!DataSet->Eof)
    {
      if (DataSet->UpdateStatus == usInserted)
      {
        DataSet->Edit();
        DataSet->FieldByName("DateCreated")->AsDateTime = Date();
        DataSet->Post();
      }
      DataSet->Next();
    }
  }
```

See Also

Resolving Update Errors On the Provider (see page 1816)

Influencing How Updates Are Applied (see page 1815)

Screening Individual Updates (see page 1817)

Applying Updates to Datasets That do Not Represent a Single Table (see page 1809)

3.2.3.11.9 Handling Server Constraints

Most relational database management systems implement constraints on their tables to enforce data integrity. A constraint is a rule that governs data values in tables and columns, or that governs data relationships across columns in different tables. For example, most SQL-92 compliant relational databases support the following constraints:

- NOT NULL, to guarantee that a value supplied to a column has a value.
- NOT NULL UNIQUE, to guarantee that column value has a value and does not duplicate any other value already in that column for another record.
- CHECK, to guarantee that a value supplied to a column falls within a certain range, or is one of a limited number of possible values.
- CONSTRAINT, a table-wide check constraint that applies to multiple columns.
- PRIMARY KEY, to designate one or more columns as the table's primary key for indexing purposes.
- FOREIGN KEY, to designate one or more columns in a table that reference another table.

Note: This list is not exclusive. Your database server may support some or all of these constraints in part or in whole, and may support additional constraints. For more information about supported constraints, see your server documentation.

Database server constraints obviously duplicate many kinds of data checks that traditional desktop database applications manage. You can take advantage of server constraints in multi-tiered database applications without having to duplicate the constraints in application server or client application code.

If the provider is working with a BDE-enabled dataset, the *Constraints* property lets you replicate and apply server constraints to data passed to and received from client datasets. When *Constraints* is *True* (the default), server constraints stored in the source dataset are included in data packets and affect client attempts to update data.

Warning: Before the provider can pass constraint information on to client datasets, it must retrieve the constraints from the database server.

There may be times when you do not want to apply server constraints to data sent to a client dataset. For example, a client dataset that receives data in packets and permits local updating of records prior to fetching more records may need to disable some server constraints that might be triggered because of the temporarily incomplete set of data. To prevent constraint replication from the provider to a client dataset, set *Constraints* to *False*. Note that client datasets can disable and enable constraints using the *DisableConstraints* and *EnableConstraints* methods. For more information about enabling and disabling constraints from the client dataset, see Handling constraints from the server (see page 1716).

See Also

Controlling What Information Is Included in Data Packets (see page 1812)

Responding to Client Data Requests (see page 1816)

Responding to Client-generated Events (see page 1817)

Responding to Client Update Requests (see page 1808)

Choosing How to Apply Updates Using a Dataset Provider (see page 1810)

Determining the Source of Data (see page 1812)

Communicating with the Client Dataset (see page 1811)

3.2.3.11.10 Influencing How Updates Are Applied

The OnUpdateData event gives your dataset provider a chance to indicate how records in the delta packet are applied to the database.

By default, changes in the delta packet are written to the database using automatically generated SQL UPDATE, INSERT, or DELETE statements such as

```
UPDATE EMPLOYEES
set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Unless you specify otherwise, all fields in the delta packet records are included in the UPDATE clause and in the WHERE clause. However, you may want to exclude some of these fields. One way to do this is to set the UpdateMode property of the provider. *UpdateMode* can be assigned any of the following values:

UpdateMode values

Value	Meaning
upWhereAll	All fields are used to locate fields (the WHERE clause).
upWhereChanged	Only key fields and fields that are changed are used to locate records.
upWhereKeyOnly	Only key fields are used to locate records.

You might, however, want even more control. For example, with the previous statement, you might want to prevent the EMPNO field from being modified by leaving it out of the UPDATE clause and leave the TITLE and DEPT fields out of the WHERE clause to avoid update conflicts when other applications have modified the data. To specify the clauses where a specific field appears, use the ProviderFlags property. *ProviderFlags* is a set that can include any of the values in the following table

ProviderFlags values

Value	Description
pfInWhere	The field appears in the WHERE clause of generated INSERT, DELETE, and UPDATE statements when *UpdateMode* is *upWhereAll* or *upWhereChanged*.
pfInUpdate	The field appears in the UPDATE clause of generated UPDATE statements.
pfInKey	The field is used in the WHERE clause of generated statements when *UpdateMode* is *upWhereKeyOnly*.
pfHidden	The field is included in records to ensure uniqueness, but can't be seen or used on the client side.

Thus, the following *OnUpdateData* event handler allows the TITLE field to be updated and uses the EMPNO and DEPT fields to locate the desired record. If an error occurs, and a second attempt is made to locate the record based only on the key, the generated SQL looks for the EMPNO field only:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet: TCustomClientDataSet);
begin
```

```
  with DataSet do
  begin
    FieldByName('TITLE').ProviderFlags := [pfInUpdate];
    FieldByName('EMPNO').ProviderFlags := [pfInWhere, pfInKey];
    FieldByName('DEPT').ProviderFlags := [pfInWhere];
  end;
end;
void __fastcall TMyDataModule1::Provider1UpdateData(TObject *Sender, TCustomClientDataSet
*DataSet)
{
  DataSet->FieldByName("EMPNO")->ProviderFlags.Clear();
  DataSet->FieldByName("EMPNO")->ProviderFlags << pfInWHere << pfInKey;
  DataSet->FieldByName("TITLE")->ProviderFlags.Clear();
  DataSet->FieldByName("TITLE")->ProviderFlags << pfInUpdate;
  DataSet->FieldByName("DEPT")->ProviderFlags.Clear();
  DataSet->FieldByName("DEPT")->ProviderFlags << pfInWhere;
}
```

Note: You can use the *UpdateFlags* property to influence how updates are applied even if you are updating to a dataset and not using dynamically generated SQL. These flags still determine which fields are used to locate records and which fields get updated.

See Also

Resolving Update Errors On the Provider (see page 1816)

Editing Delta Packets Before Updating the Database (see page 1813)

Screening Individual Updates (see page 1817)

Applying Updates to Datasets That do Not Represent a Single Table (see page 1809)

3.2.3.11.11 Resolving Update Errors On the Provider

When an error condition arises as the dataset provider tries to post a record in the delta packet, an OnUpdateError event occurs. If the provider can't resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the provider returns a count of the errors it encountered, and copies the unresolved records into a results data packet that it passes back to the client for further reconciliation.

In multi-tiered applications, this mechanism lets you handle any update errors you can resolve mechanically on the application server, while still allowing user interaction on the client application to correct error conditions.

The *OnUpdateError* handler gets a copy of the record that could not be changed, an error code from the database, and an indication of whether the resolver was trying to insert, delete, or update the record. The problem record is passed back in a client dataset. You should never use the data navigation methods on this dataset. However, for each field in the dataset, you can use the NewValue, OldValue, and CurValue properties to determine the cause of the problem and make any modifications to resolve the update error. If the *OnUpdateError* event handler can correct the problem, it sets the *Response* parameter so that the corrected record is applied.

See Also

Reconciling Update Errors (see page 1737)

Screening Individual Updates (see page 1817)

Applying Updates to Datasets That do Not Represent a Single Table (see page 1809)

3.2.3.11.12 Responding to Client Data Requests

Usually client requests for data are handled automatically. A client dataset or XML broker requests a data packet by calling GetRecords (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the

associated dataset or XML document, creating a data packet, and sending the packet to the client.

The provider has the option of editing data after it has been assembled into a data packet but before the packet is sent to the client. For example, you might want to remove records from the packet based on some criterion (such as the user's level of access), or, in a multi-tiered application, you might want to encrypt sensitive data before it is sent on to the client.

To edit the data packet before sending it on to the client, write an OnGetData event handler. *OnGetData* event handlers provide the data packet as a parameter in the form of a client dataset. Using the methods of this client dataset, you can edit data before it is sent to the client.

As with all method calls made through the *IAppServer* interface, the provider can communicate persistent state information (see page 1540) with a client dataset before and after the call to *GetRecords*. This communication takes place using the BeforeGetRecords and AfterGetRecords event handlers.

See Also

Controlling What Information Is Included in Data Packets (see page 1812)

Responding to Client-generated Events (see page 1817)

Handling Server Constraints (see page 1814)

Responding to Client Update Requests (see page 1808)

Choosing How to Apply Updates Using a Dataset Provider (see page 1810)

Determining the Source of Data (see page 1812)

Communicating with the Client Dataset (see page 1811)

3.2.3.11.13 Responding to Client-generated Events

Provider components implement a general-purpose event that lets you create your own calls from client datasets directly to the provider. This is the OnDataRequest event.

OnDataRequest is not part of the normal functioning of the provider. It is simply a hook to allow your client datasets to communicate directly with providers. The event handler takes an OleVariant as an input parameter and returns an OleVariant. By using OleVariants, the interface is sufficiently general to accommodate almost any information you want to pass to or from the provider.

To generate an *OnDataRequest* event, the client application calls the DataRequest method of the client dataset.

See Also

Controlling What Information Is Included in Data Packets (see page 1812)

Responding to Client Data Requests (see page 1816)

Handling Server Constraints (see page 1814)

Responding to Client Update Requests (see page 1808)

Choosing How to Apply Updates Using a Dataset Provider (see page 1810)

Determining the Source of Data (see page 1812)

Communicating with the Client Dataset (see page 1811)

3.2.3.11.14 Screening Individual Updates

Immediately before each update is applied, a dataset provider receives a BeforeUpdateRecord event. You can use this event to

edit records before they are applied, similar to the way you can use the *OnUpdateData* event to edit entire delta packets (see page 1813). For example, the provider does not compare BLOB fields (such as memos) when checking for update conflicts. If you want to check for update errors involving BLOB fields, you can use the *BeforeUpdateRecord* event.

In addition, you can use this event to apply updates yourself or to screen and reject updates. The *BeforeUpdateRecord* event handler lets you signal that an update has been handled already and should not be applied. The provider then skips that record, but does not count it as an update error. For example, this event provides a mechanism for applying updates to a stored procedure (which can't be updated automatically), allowing the provider to skip any automatic processing once the record is updated from within the event handler.

See Also

Resolving Update Errors On the Provider (see page 1816)

Editing Delta Packets Before Updating the Database (see page 1813)

Influencing How Updates Are Applied (see page 1815)

Applying Updates to Datasets That do Not Represent a Single Table (see page 1809)

3.2.3.11.15 Setting Options That Influence the Data Packets

The Options property of a dataset provider lets you specify when BLOBs or nested detail tables are sent, whether field display properties are included, what type of updates are allowed, and so on. The following table lists the possible values that can be included in *Options*.

Provider options

Value	Meaning
poAutoRefresh	The provider refreshes the client dataset with current record values whenever it applies updates.
poReadOnly	The client dataset can't apply updates to the provider.
poDisableEdits	Client datasets can't modify existing data values. If the user tries to edit a field, the client dataset raises exception. (This does not affect the client dataset's ability to insert or delete records).
poDisableInserts	Client datasets can't insert new records. If the user tries to insert a new record, the client dataset raises an exception. (This does not affect the client dataset's ability to delete records or modify existing data)
poDisableDeletes	Client datasets can't delete records. If the user tries to delete a record, the client dataset raises an exception. (This does not affect the client dataset's ability to insert or modify records)
poFetchBlobsOnDemand	BLOB field values are not included in data packets. Instead, client datasets must request these values on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchBlobs* method to retrieve BLOB data.
poFetchDetailsOnDemand	When the provider's dataset represents the master of a master/detail relationship, nested detail values are not included in data packets. Instead, client datasets request these on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, it requests these values automatically. Otherwise, the application must call the client dataset's *FetchDetails* method to retrieve nested details.
poIncFieldProps	The data packet includes the following field properties (where applicable): *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask*, *DisplayValues.*
poCascadeDeletes	When the provider's dataset represents the master of a master/detail relationship, the server automatically deletes detail records when master records are deleted. To use this option, the database server must be set up to perform cascaded deletes as part of its referential integrity.

poCascadeUpdates	When the provider's dataset represents the master of a master/detail relationship, key values on detail tables are updated automatically when the corresponding values are changed in master records. To use this option, the database server must be set up to perform cascaded updates as part of its referential integrity.
poAllowMultiRecordUpdates	A single update can cause more than one record of the underlying database table to change. This can be the result of triggers, referential integrity, SQL statements on the source dataset, and so on. Note that if an error occurs, the event handlers provide access to the record that was updated, not the other records that change in consequence.
poNoReset	Client datasets can't specify that the provider should reposition the cursor on the first record before providing data.
poPropogateChanges	Changes made by the server to updated records as part of the update process are sent back to the client and merged into the client dataset.
poAllowCommandText	The client can override the associated dataset's SQL text or the name of the table or stored procedure it represents.
poRetainServerOrder	The client dataset should not re-sort the records in the dataset to enforce a default order.

See Also

Specifying What Fields Appear in Data Packets (see page 1819)

Adding Custom Information to Data Packets (see page 1808)

3.2.3.11.16 Specifying What Fields Appear in Data Packets

When using a dataset provider, you can control what fields are included in data packets by creating persistent fields (see page 1859) on the dataset that the provider uses to build data packets. The provider then includes only these fields. Fields whose values are generated dynamically by the source dataset (such as calculated fields or lookup fields) can be included, but appear to client datasets on the receiving end as static read-only fields.

If the client dataset will be editing the data and applying updates, you must include enough fields so that there are no duplicate records in the data packet. Otherwise, when the updates are applied, it is impossible to determine which record to update. If you do not want the client dataset to be able to see or use extra fields provided only to ensure uniqueness, set the ProviderFlags property for those fields to include *pfHidden*.

Note: Including enough fields to avoid duplicate records is also a consideration when the provider's source dataset represents a query. You must specify the query so that it includes enough fields to ensure all records are unique, even if your application does not use all the fields.

See Also

Setting Options That Influence the Data Packets (see page 1818)

Adding Custom Information to Data Packets (see page 1808)

Persistent Field Components (see page 1859)

3.2.3.11.17 Using Provider Components

Provider components (TDataSetProvider and TXMLTransformProvider) supply the most common mechanism by which client datasets obtain their data. Providers

- Receive data requests from a client dataset (or XML broker), fetch the requested data, package the data into a transportable data packet, and return the data to the client dataset (or XML broker). This activity is called "providing."
- Receive updated data from a client dataset (or XML broker), apply updates to the database server, source dataset, or source

XML document, and log any updates that cannot be applied, returning unresolved updates to the client dataset for further reconciliation. This activity is called "resolving."

Most of the work of a provider component happens automatically. You need not write any code on the provider to create data packets from the data in a dataset or XML document or to apply updates. However, provider components include a number of events and properties that allow your application more direct control over what information is packaged for clients and how your application responds to client requests.

When using *TBDEClientDataSet*, *TSimpleDataSet*, or *TIBClientDataSet*, the provider is internal to the client dataset, and the application has no direct access to it. When using *TClientDataSet* or *TXMLBroker*, however, the provider is a separate component that you can use to control what information is packaged for clients and for responding to events that occur around the process of providing and resolving. The client datasets that have internal providers surface some of the internal provider's properties and events as their own properties and events, but for the greatest amount of control, you may want to use *TClientDataSet* with a separate provider component.

When using a separate provider component, it can reside in the same application as the client dataset (or XML broker), or it can reside on an application server as part of a multi-tiered application.

The following topics describe how to use a provider component to control the interaction with client datasets or XML brokers.

- Determining the Source of Data (see page 1812)
- Communicating with the Client Dataset (see page 1811)
- Choosing How to Apply Updates Using a Dataset Provider (see page 1810)
- Controlling what Information is Included in Data Packets (see page 1812)
- Responding to Client Data Requests (see page 1816)
- Responding to Client Update Requests (see page 1808)
- Responding to Client-generated Events (see page 1817)
- Handling Server Constraints (see page 1814)

See Also

Creating MultiTiered Applications (see page 1518)

Creating and Using a Client Dataset (see page 1740)

Designing Database Applications (see page 1566)

Using Data Controls (see page 1778)

Working with Field Components (see page 1877)

Connecting to Databases (see page 1506)

Working With ADO Components (see page 1494)

Using Unidirectional Datasets (see page 1823)

Using the Borland Database Engine (see page 1643)

Writing Web-based Client Applications (see page 1551)

3.2.3.12 Using dbExpress Components

Topics

Name	Description
Using dbExpress Datasets (see page 1823)	*dbExpress* is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfaces. When you deploy a database application that uses *dbExpress*, you need only include a dll (the server-specific driver) with the application files you build. *dbExpress* lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command... more (see page 1823)
Accessing dbExpress Schema Information (see page 1824)	There are two ways to obtain information about what is available on the server. This information, called schema information or metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses). The simplest way to obtain this metadata is to use the methods of TSQLConnection. These methods fill an existing string list or list object with the names of tables, stored procedures, fields, or indexes, or with parameter descriptors. This... more (see page 1824)
Connecting to the Database Server (see page 1825)	The first step when working with a dbExpress dataset is to connect it to a database server. At design time, once a dataset has an active connection to a database server, the **Object Inspector** can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the **Object Inspector** can list what stored procedures are available on the server. The connection to a database server is represented by a separate TSQLConnection component. You work with *TSQLConnection* like any other database connection component (see page 1506). To use *TSQLConnection* to connect a... more (see page 1825)
Creating and Modifying Server Metadata (see page 1826)	Most of the commands that do not return data fall into two categories: those that you use to edit data (such as INSERT, DELETE, and UPDATE commands), and those that you use to create or modify entities on the server such as tables, indexes, and stored procedures. If you don't want to use explicit SQL commands for editing, you can link your unidirectional dataset to a client dataset (see page 1559) and let it handle all the generation of all SQL commands concerned with editing. In fact, this is the recommended approach because data-aware controls are designed to perform edits through a dataset... more (see page 1826)
Debugging dbExpress Applications (see page 1827)	While you are debugging your database application, it may prove useful to monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the *dbExpress* driver).
Executing Commands That Do Not Return Records (see page 1829)	You can use a dbExpress dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language. The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it. They merely pass... more (see page 1829)
Executing the Command (see page 1829)	To execute a query or stored procedure that does not return any records, you do not use the *Active* property or the *Open* method. Instead, you must use The ExecSQL method if the dataset is an instance of *TSQLDataSet* or *TSQLQuery*.

Fetching Metadata into a dbExpress Dataset (see page 1830)	To populate a unidirectional dataset with metadata from the database server, you must first indicate what data you want to see, using the SetSchemaInfo method. *SetSchemaInfo* takes three parameters: • The type of schema information (metadata) you want to fetch. This can be a list of tables (*stTables*), a list of system tables (*stSysTables*), a list of stored procedures (*stProcedures*), a list of fields in a table (*stColumns*), a list of indexes (*stIndexes*), or a list of parameters used by a stored procedure (*stProcedureParams*). Each type of information uses... more (see page 1830)
Fetching the Data (see page 1830)	Once you have specified the source of the data (see page 1836), you must fetch the data before your application can access it. Once the dataset has fetched the data, data-aware controls linked to the dataset through a data source automatically display data values and client datasets linked to the dataset through a provider can be populated with records. As with any dataset, there are two ways to fetch the data for a dbExpress dataset: One way is to set the Active*property to True, either at design time in the* ***Object Inspector****, or in code at runtime:*
Representing the Records in a Table (see page 1832)	When you want to represent all of the fields and all of the records in a single underlying database table, you can use either *TSQLDataSet* or *TSQLTable* to generate the query for you rather than writing the SQL yourself. **Note:** If server performance is a concern, you may want to compose the query explicitly rather than relying on an automatically-generated query. Automatically-generated queries use wildcards rather than explicitly listing all of the fields in the table. This can result in slightly slower performance on the server. The wildcard (*) in automatically-generated queries is more robust to changes in the fields... more (see page 1832)
Representing the Results of a Query (see page 1832)	Using a query is the most general way to specify a set of records. Queries are simply commands written in SQL. You can use either *TSQLDataSet* or *TSQLQuery* to represent the result of a query. When using *TSQLDataSet*, set the CommandType property to *ctQuery* and assign the text of the query statement to the CommandText property. When using *TSQLQuery*, assign the query to the SQL property instead. These properties work the same way for all general-purpose or query-type datasets. Specifying the query (see page 1613) discusses them in greater detail. When you specify the query, it can include parameters, or variables,... more (see page 1832)
Representing the Results of a Stored Procedure (see page 1833)	Stored procedures are sets of SQL statements that are named and stored on an SQL server. How you indicate the stored procedure you want to execute depends on the type of unidirectional dataset you are using. When using TSQLDataSet, to specify a stored procedure: • Set the CommandType property to *ctStoredProc*. • Specify the name of the stored procedure as the value of the CommandText property:
Setting Up Master/Detail Linked Relationships (see page 1833)	There are two ways to use linked datasets to set up a master/detail relationship with a dbExpress dataset as the detail set. Which method you use depends on the type of unidirectional dataset (see page 1836) you are using. Once you have set up such a relationship, the unidirectional dataset (the "many" in a one-to-many relationship) provides access only to those records that correspond to the current record on the master set (the "one" in the one-to-many relationship). *TSQLDataSet* and *TSQLQuery* require you to use a parameterized query to establish a master/detail relationship. This is the technique for creating such relationships on all... more (see page 1833)
Setting Up TSQLConnection (see page 1834)	In order to describe a database connection in sufficient detail for TSQLConnection to open a connection, you must identify both the driver to use and a set of connection parameters the are passed to that driver.

Specifying the Command to Execute (see page 1835)	With unidirectional datasets, the way you specify the command to execute is the same whether the command results in a dataset or not. That is: When using TSQLDataSet, use the CommandType and CommandText properties to specify the command: • If *CommandType* is *ctQuery*, *CommandText* is the SQL statement to pass to the server. • If *CommandType* is *ctStoredProc*, *CommandText* is the name of a stored procedure to execute. When using TSQLQuery, use the SQL property to specify the SQL statement to pass to the server. When using TSQLStoredProc, use the StoredProcName property to specify the name of the stored procedure... more (see page 1835)
Specifying What Data to Display (see page 1836)	There are a number of ways to specify what data a dbExpress dataset represents. Which method you choose depends on the type of unidirectional dataset (see page 1836) you are using and whether the information comes from a single database table, the results of a query, or from a stored procedure. When you work with a TSQLDataSet component, use the CommandType property to indicate where the dataset gets its data. *CommandType* can take any of the following values: • *ctQuery*: When *CommandType* is *ctQuery*, *TSQLDataSet* executes a query you specify. If the query is a SELECT command, the dataset contains the resulting... more (see page 1836)
Types of dbExpress Datasets (see page 1836)	The *dbExpress* category of the **Tool palette** contains four types of unidirectional dataset: TSQLDataSet, TSQLQuery, TSQLTable, and TSQLStoredProc. *TSQLDataSet* is the most general of the four. You can use an SQL dataset to represent any data available through *dbExpress*, or to send commands to a database accessed through *dbExpress*. This is the recommended component to use for working with database tables in new database applications. *TSQLQuery* is a query-type dataset (see page 1618) that encapsulates an SQL statement and enables applications to access the resulting records, if any. *TSQLTable* is a table-type dataset (see page 1620) that represents all of the rows and columns... more (see page 1836)
The Structure of Metadata Datasets (see page 1837)	For each type of metadata you can access using *TSQLDataSet*, there is a predefined set of columns (fields) that are populated with information about the items of the requested type.

3.2.3.12.1 Using dbExpress Datasets

dbExpress is a set of lightweight database drivers that provide fast access to SQL database servers. For each supported database, *dbExpress* provides a driver that adapts the server-specific software to a set of uniform *dbExpress* interfaces. When you deploy a database application that uses *dbExpress*, you need only include a dll (the server-specific driver) with the application files you build.

dbExpress lets you access databases using unidirectional datasets. Unidirectional datasets are designed for quick lightweight access to database information, with minimal overhead. Like other datasets, they can send an SQL command to the database server, and if the command returns a set of records, obtain a cursor for accessing those records. However, unidirectional datasets can only retrieve a unidirectional cursor. They do not buffer data in memory, which makes them faster and less resource-intensive than other types of dataset. However, because there are no buffered records, unidirectional datasets are also less flexible than other datasets. Many of the capabilities introduced by TDataSet (see page 1632) are either unimplemented in unidirectional datasets, or cause them to raise exceptions. For example:

- The only supported navigation methods are the First and Next methods. Most others raise exceptions. Some, such as the methods involved in bookmark support, simply do nothing.
- There is no built-in support for editing because editing requires a buffer to hold the edits. The CanModify property is always *False*, so attempts to put the dataset into edit mode always fail. You can, however, use unidirectional datasets to update data

using an SQL UPDATE command or provide conventional editing support by using a dbExpress-enabled client dataset or connecting the dataset to a client dataset (see page 1561) .

- There is no support for filters, because filters work with multiple records, which requires buffering. If you try to filter a unidirectional dataset, it raises an exception. Instead, all limits on what data appears must be imposed using the SQL command that defines the data for the dataset.
- There is no support for lookup fields, which require buffering to hold multiple records containing lookup values. If you define a lookup field on a unidirectional dataset, it does not work properly.

Despite these limitations, unidirectional datasets are a powerful way to access data. They are the fastest data access mechanism, and very simple to use and deploy.

The following topics describe unidirectional datasets in greater detail:

- Types of dbExpress datasets (see page 1836)
- Connecting to the database server (see page 1825)
- Specifying what data to display (see page 1836)
- Fetching the data (see page 1830)
- Executing commands that do not return records (see page 1829)
- Setting up master/detail linked cursors (see page 1833)
- Accessing Schema Information (see page 1824)
- Debugging dbExpress applications (see page 1827)

See Also

Understanding Datasets (see page 1632)

Working with Field Components (see page 1877)

Connecting to Databases (see page 1506)

Creating and Using a Client Dataset (see page 1740)

Designing Database Applications (see page 1566)

Using Data Controls (see page 1778)

Working with ADO Components (see page 1494)

Using the Borland Database Engine (see page 1643)

Using Provider Components (see page 1819)

Creating Multi-tiered Applications: Overview (see page 1518)

Using Decision Support Components (see page 1800)

Using XML in Database Applications (see page 1847)

Configuring TSQL Connection (see page 53)

Using Data Explorer to get Connection Information (see page 52)

3.2.3.12.2 Accessing dbExpress Schema Information

There are two ways to obtain information about what is available on the server. This information, called schema information or metadata, includes information about what tables and stored procedures are available on the server and information about these tables and stored procedures (such as the fields a table contains, the indexes that are defined, and the parameters a stored procedure uses).

The simplest way to obtain this metadata is to use the methods of TSQLConnection. These methods fill an existing string list or list object with the names of tables, stored procedures, fields, or indexes, or with parameter descriptors. This technique is the same as the way you fill lists with metadata for any other database connection component. These methods are described in Obtaining metadata (see page 1501).

If you require more detailed schema information, you can populate a unidirectional dataset with metadata. Instead of a simple list, the unidirectional dataset is filled with schema information, where each record represents a single table, stored procedure, index, field, or parameter. See Fetching metadata into a unidirectional dataset (see page 1830) for details on populating a unidirectional dataset with schema information.

See Also

Connecting to the Database Server (see page 1825)

Types of dbExpress Datasets (see page 1836)

Specifying What Data to Display (see page 1836)

Executing Commands That Do Not Return Records (see page 1829)

Fetching the Data (see page 1830)

Debugging dbExpress Applications (see page 55)

3.2.3.12.3 Connecting to the Database Server

The first step when working with a dbExpress dataset is to connect it to a database server. At design time, once a dataset has an active connection to a database server, the **Object Inspector** can provide drop-down lists of values for other properties. For example, when representing a stored procedure, you must have an active connection before the **Object Inspector** can list what stored procedures are available on the server.

The connection to a database server is represented by a separate TSQLConnection component. You work with *TSQLConnection* like any other database connection component (see page 1506).

To use *TSQLConnection* to connect a unidirectional dataset to a database server, set the SQLConnection property. At design time, you can choose the SQL connection component from a drop-down list in the **Object Inspector**. If you make this assignment at runtime, be sure that the connection is active:

```
SQLDataSet1.SQLConnection := SQLConnection1;
SQLConnection1.Connected := True;
SQLDataSet1->SQLConnection = SQLConnection1;
SQLConnection1->Connected = true;
```

Typically, all unidirectional datasets in an application share the same connection component, unless you are working with data from multiple database servers. However, you may want to use a separate connection for each dataset if the server does not support multiple statements per connection. Check whether the database server requires a separate connection for each dataset by reading the MaxStmtsPerConn property. By default, *TSQLConnection* generates connections as needed when the server limits the number of statements that can be executed over a connection. If you want to keep stricter track of the connections you are using, set the AutoClone property to *False*.

Before you assign the *SQLConnection* property, you will need to set up the TSQLConnection component (see page 1834) so that it identifies the database server and any required connection parameters (including which database to use on the server, the host name of the machine running the server, the username, password, and so on).

See Also

Specifying What Data to Display (see page 1836)

Types of dbExpress Datasets (see page 1836)

Fetching the Data (see page 1830)

Executing Commands That Do Not Return Records (see page 1829)

Accessing Schema Information (see page 1824)

Debugging dbExpress Applications (see page 55)

3.2.3.12.4 Creating and Modifying Server Metadata

Most of the commands that do not return data fall into two categories: those that you use to edit data (such as INSERT, DELETE, and UPDATE commands), and those that you use to create or modify entities on the server such as tables, indexes, and stored procedures.

If you don't want to use explicit SQL commands for editing, you can link your unidirectional dataset to a client dataset (see page 1559) and let it handle all the generation of all SQL commands concerned with editing. In fact, this is the recommended approach because data-aware controls are designed to perform edits through a dataset such as *TClientDataSet*.

The only way your application can create or modify metadata on the server, however, is to send a command. Not all database drivers support the same SQL syntax. It is beyond the scope of this topic to describe the SQL syntax supported by each database type and the differences between the database types. For a comprehensive and up-to-date discussion of the SQL implementation for a given database system, see the documentation that comes with that system.

In general, use the CREATE TABLE statement to create tables in a database and CREATE INDEX to create new indexes for those tables. Where supported, use other CREATE statements for adding various metadata objects, such as CREATE DOMAIN, CREATE VIEW, CREATE SCHEMA, and CREATE PROCEDURE.

For each of the CREATE statements, there is a corresponding DROP statement to delete the metadata object. These statements include DROP TABLE, DROP VIEW, DROP DOMAIN, DROP SCHEMA, and DROP PROCEDURE.

To change the structure of a table, use the ALTER TABLE statement. ALTER TABLE has ADD and DROP clauses to create new elements in a table and to delete them. For example, use the ADD COLUMN clause to add a new column to the table and DROP CONSTRAINT to delete an existing constraint from the table.

For example, the following statement creates a stored procedure called GET_EMP_PROJ on an InterBase database:

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The following code uses a *TSQLDataSet* to create this stored procedure. Note the use of the *ParamCheck* property to prevent the dataset from confusing the parameters in the stored procedure definition (:EMP_NO and :PROJ_ID) with a parameter of the query that creates the stored procedure.

```
with SQLDataSet1 do
begin
  ParamCheck := False;
  CommandType := ctQuery;
  CommandText := 'CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) ' +
      'RETURNS (PROJ_ID CHAR(5)) AS ' +
      'BEGIN ' +
        'FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT ' +
        'WHERE EMP_NO = :EMP_NO ' +
        'INTO :PROJ_ID ' +
          'DO SUSPEND; ' +
```

```
        END';
    ExecSQL;
end;
SQLDataSet1->ParamCheck = false;
SQLDataSet1->CommandType = ctQuery;
SQLDataSet1->CommandText = "CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT) RETURNS (PROJ_ID
CHAR(5)) AS BEGIN FOR SELECT PROJ_ID FROM EMPLOYEE_PROJECT WHERE EMP_NO = :EMP_NO INTO
:PROJ_ID DO SUSPEND; END";
SQLDataSet1->ExecSQL();
```

See Also

Specifying the Command to Execute (see page 1835)

Executing the Command (see page 1829)

3.2.3.12.5 Debugging dbExpress Applications

While you are debugging your database application, it may prove useful to monitor the SQL messages that are sent to and from the database server through your connection component, including those that are generated automatically for you (for example by a provider component or by the *dbExpress* driver).

Using TSQLMonitor to monitor SQL commands

TSQLConnection uses a companion component, TSQLMonitor, to intercept these messages and save them in a string list. *TSQLMonitor* works much like the SQL monitor utility that you can use with the BDE, except that it monitors only those commands involving a single *TSQLConnection* component rather than all commands managed by *dbExpress*.

To use TSQLMonitor

1. Add a *TSQLMonitor* component to the form or data module containing the *TSQLConnection* component whose SQL commands you want to monitor.
2. Set its SQLConnection property to the *TSQLConnection* component.
3. Set the SQL monitor's Active property to *True*.

Flags for monitoring SQL commands

Flag	Meaning
traceUNKNOWN	All SQL commands.
traceQPREPARE	prepared queries sent to the server.
traceQEXECUTE	Queries to be executed by the server. Note that a single statement may be prepared once and executed several times with different parameter bindings.
traceERROR	Error messages returned by the server. The error message may include an error code, depending on the server.
traceSTMT	Operations to be performed such as ALLOCATE, PREPARE, EXECUTE, and FETCH.
traceCONNECT	Operations associated with connecting and disconnecting to databases, including allocation of connection handles and freeing connection handles, if required by server.
traceTRANSACT	Transaction operations such as BEGIN, COMMIT, and ROLLBACK (ABORT).
traceBLOB	Operations on Binary Large Object (BLOB) data, including STORE BLOB, GET BLOB HANDLE, and so on.
traceMISC	commands not covered by any other flag.
traceVENDOR	API function calls to the server. For example, ORLON for Oracle, ISC_ATTACH for InterBase.
traceDATAIN	Parameter data sent to servers when doing INSERTs or UPDATEs.
traceDATAOUT	Data retrieved from servers.

As SQL commands are sent to the server, the SQL monitor's TraceList property is automatically updated to list all the SQL commands that are intercepted.

You can save this list to a file by specifying a value for the FileName property and then setting the AutoSave property to *True*. *AutoSave* causes the SQL monitor to save the contents of the *TraceList* property to a file every time is logs a new message.

If you do not want the overhead of saving a file every time a message is logged, you can use the OnLogTrace event handler to only save files after a number of messages have been logged. For example, the following event handler saves the contents of *TraceList* every 10th message, clearing the log after saving it so that the list never gets too long:

```
procedure TForm1.SQLMonitor1LogTrace(Sender: TObject; CBInfo: Pointer);
var
  LogFileName: string;
begin
  with Sender as TSQLMonitor do
  begin
    if TraceCount = 10 then
    begin
      LogFileName := 'c:\log' + IntToStr(Tag) + '.txt';
      Tag := Tag + 1; {ensure next log file has a different name }
      SaveToFile(LogFileName);
      TraceList.Clear; { clear list }
    end;
  end;
end;
void __fastcall TForm1::SQLMonitor1LogTrace(TObject *Sender, void *CBInfo)
{
  TSQLMonitor *pMonitor = dynamic_cast<TSQLMonitor *>(Sender);
  if (pMonitor->TraceCount == 10)
  {
    // build unique file name
    AnsiString LogFileName = "c:\\log";
    LogFileName = LogFileName + IntToStr(pMonitor->Tag);
    LogFileName = LogFileName + ".txt"
    pMonitor->Tag = pMonitor->Tag + 1;
    // Save contents of log and clear the list
    pMonitor->SaveToFile(LogFileName);
    pMonitor->TraceList->Clear();

}
```

Note: If you were to use the previous event handler, you would also want to save any partial list (fewer than 10 entries) when the application shuts down.

Using a callback to monitor SQL commands

Instead of using *TSQLMonitor*, you can customize the way your application traces SQL commands by using the SQL connection component's SetTraceEvent method. *SetTraceEvent* takes a TDBXTraceEvent parameter .

The *dbExpress* driver triggers the event every time the SQL connection component passes a command to the server or the server returns an error message.

Warning: Do not call *SetTraceEvent* if the *TSQLConnection* object has an associated *TSQLMonitor* component. *TSQLMonitor* uses the callback mechanism to work, and *TSQLConnection* can only support one callback at a time.

See Also

Connecting to the Database Server (see page 1825)

Types of Unidirectional Datasets (see page 1836)

Specifying What Data to Display (see page 1836)

Executing Commands That Do Not Return Records (see page 1829)

Fetching the Data (see page 1830)

Accessing Schema Information (see page 51)

3.2.3.12.6 Executing Commands That Do Not Return Records

You can use a dbExpress dataset even if the query or stored procedure it represents does not return any records. Such commands include statements that use Data Definition Language (DDL) or Data Manipulation Language (DML) statements other than SELECT statements (For example, INSERT, DELETE, UPDATE, CREATE INDEX, and ALTER TABLE commands do not return any records). The language used in commands is server-specific, but usually compliant with the SQL-92 standard for the SQL language.

The SQL command you execute must be acceptable to the server you are using. Unidirectional datasets neither evaluate the SQL nor execute it. They merely pass the command to the server for execution.

Note: If the command does not return any records, you do not need to use a unidirectional dataset at all, because there is no need for the dataset methods that provide access to a set of records. The SQL connection component that connects to the database server can be used directly to execute a command on the server. See Sending commands to the server (see page 1502) for details.

The following topics discuss how to create and execute a command that does not return any records:

- Specifying the command to execute (see page 1835)
- Executing the command (see page 1829)

In addition, the topic,Creating and modifying server metadata (see page 1826), discusses some of the SQL commands that do not return datasets:

See Also

Connecting to the Database Server (see page 1825)

Types of dbExpress Datasets (see page 1836)

Specifying What Data to Display (see page 1836)

Creating Master/detail Relationships (see page 1590)

Accessing Schema Information (see page 51)

Fetching the Data (see page 1830)

Debugging dbExpress Applications (see page 55)

3.2.3.12.7 Executing the Command

To execute a query or stored procedure that does not return any records, you do not use the *Active* property or the *Open* method. Instead, you must use

The ExecSQL method if the dataset is an instance of *TSQLDataSet* or *TSQLQuery*.

```
FixTicket.CommandText := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
FixTicket.ExecSQL;
FixTicket->CommandText = "DELETE FROM TrafficViolations WHERE (TicketID = 1099)";
FixTicket->ExecSQL();
```

The ExecProc method if the dataset is an instance of *TSQLStoredProc*.

```
SQLStoredProc1.StoredProcName := 'MyCommandWithNoResults';
SQLStoredProc1.ExecProc;
SQLStoredProc1->StoredProcName = "MyCommandWithNoResults";
SQLStoredProc1->ExecProc();
```

If you are executing the query or stored procedure multiple times, it is a good idea to set the Prepared property to *True*.

See Also

Specifying the Command to Execute (see page 1835)

Creating and Modifying Server Metadata (see page 1826)

3.2.3.12.8 Fetching Metadata into a dbExpress Dataset

To populate a unidirectional dataset with metadata from the database server, you must first indicate what data you want to see, using the SetSchemaInfo method. *SetSchemaInfo* takes three parameters:

- The type of schema information (metadata) you want to fetch. This can be a list of tables (*stTables*), a list of system tables (*stSysTables*), a list of stored procedures (*stProcedures*), a list of fields in a table (*stColumns*), a list of indexes (*stIndexes*), or a list of parameters used by a stored procedure (*stProcedureParams*). Each type of information uses a different set of fields to describe the items in the list. For details on the structures of these datasets, see The structure of metadata datasets (see page 1837).
- If you are fetching information about fields, indexes, or stored procedure parameters, the name of the table or stored procedure to which they apply. If you are fetching any other type of schema information, this parameter is nil.
- A pattern that must be matched for every name returned. This pattern is an SQL pattern such as 'Cust%', which uses the wildcards '%' (to match a string of arbitrary characters of any length) and '_' (to match a single arbitrary character). To use a literal percent or underscore in a pattern, the character is doubled (%% or __). If you do not want to use a pattern, this parameter can be nil.

If you are fetching schema information about tables (*stTables*), the resulting schema information can describe ordinary tables, system tables, views, and/or synonyms, depending on the value of the SQL connection's TableScope property.

The following call requests a table listing all system tables (server tables that contain metadata):

```
SQLDataSet1.SetSchemaInfo(stSysTable, "", "");
SQLDataSet1->SetSchemaInfo(stSysTable, "", "");
```

When you open the dataset after this call to *SetSchemaInfo*, the resulting dataset has a record for each table, with columns giving the table name, type, schema name, and so on. If the server does not use system tables to store metadata (for example MySQL), when you open the dataset it contains no records.

The previous example used only the first parameter. Suppose, Instead, you want to obtain a list of input parameters for a stored procedure named 'MyProc'. Suppose, further, that the person who wrote that stored procedure named all parameters using a prefix to indicate whether they were input or output parameters ('inName', 'outValue' and so on). You could call *SetSchemaInfo* as follows:

```
SQLDataSet1.SetSchemaInfo(stProcedureParams, "MyProc", "in%");
SQLDataSet1->SetSchemaInfo(stProcedureParams, "MyProc", "in%");
```

The resulting dataset is a table of input parameters with columns to describe the properties of each parameter.

Fetching data after using the dataset for metadata

There are two ways to return to executing queries or stored procedures with the dataset after a call to *SetSchemaInfo*:

- Change the CommandText property, specifying the query, table, or stored procedure from which you want to fetch data.
- Call *SetSchemaInfo*, setting the first parameter to *stNoSchema*. In this case, the dataset reverts to fetching the data specified by the current value of *CommandText*.

See Also

Obtaining Metadata (see page 1501)

3.2.3.12.9 Fetching the Data

Once you have specified the source of the data (see page 1836), you must fetch the data before your application can access

it. Once the dataset has fetched the data, data-aware controls linked to the dataset through a data source automatically display data values and client datasets linked to the dataset through a provider can be populated with records.

As with any dataset, there are two ways to fetch the data for a dbExpress dataset:

One way is to set the Active*property to True, either at design time in the* ***Object Inspector****, or in code at runtime:*

```
CustQuery.Active := True;
CustQuery->Active = true;
```

Another way is to call the Open method at runtime,

```
CustQuery.Open;
CustQuery->Open();
```

Use the *Active* property or the *Open* method with any unidirectional dataset that obtains records from the server. It does not matter whether these records come from a SELECT query (including automatically-generated queries when the *CommandType* is *ctTable*) or a stored procedure.

Preparing the dataset

Before a query or stored procedure can execute on the server, it must first be "prepared". Preparing the dataset means that *dbExpress* and the server allocate resources for the statement and its parameters. If *CommandType* is *ctTable*, this is when the dataset generates its SELECT query. Any parameters that are not bound by the server are folded into a query at this point.

Unidirectional datasets are automatically prepared when you set *Active* to *True* or call the *Open* method. When you close the dataset, the resources allocated for executing the statement are freed. If you intend to execute the query or stored procedure more than once, you can improve performance by explicitly preparing the dataset before you open it the first time. To explicitly prepare a dataset, set its Prepared property to *True*.

```
CustQuery.Prepared := True;
CustQuery->Prepared = true;
```

When you explicitly prepare the dataset, the resources allocated for executing the statement are not freed until you set *Prepared* to *False*.

Set the *Prepared* property to *False* if you want to ensure that the dataset is re-prepared before it executes (for example, if you change a parameter value or the *SortFieldNames* property).

Fetching multiple datasets

Some stored procedures return multiple sets of records. The dataset only fetches the first set when you open it. In order to access the other sets of records, call the NextRecordSet method:

```
var
  DataSet2: TCustomSQLDataSet;
  nRows: Integer;
begin
  DataSet2 := SQLStoredProc1.NextRecordSet;
  ...
TCustomSQLDataSet *DataSet2 = SQLStoredProc1->NextRecordSet();
```

NextRecordSet returns a newly created *TCustomSQLDataSet* component that provides access to the next set of records. That is, the first time you call *NextRecordSet*, it returns a dataset for the second set of records. Calling *NextRecordSet* returns a third dataset, and so on, until there are no more sets of records. When there are no additional datasets, *NextRecordSet* returns **nil**.

See Also

Connecting to the Database Server (see page 1825)

Types of dbExpress Datasets (see page 1836)

Specifying What Data to Display (see page 1836)

Executing Commands That Do Not Return Records (see page 1829)

Accessing Schema Information (see page 1824)

Opening and Closing Datasets (see page 1604)

Debugging dbExpress Applications (see page 1827)

3.2.3.12.10 Representing the Records in a Table

When you want to represent all of the fields and all of the records in a single underlying database table, you can use either *TSQLDataSet* or *TSQLTable* to generate the query for you rather than writing the SQL yourself.

Note: If server performance is a concern, you may want to compose the query explicitly rather than relying on an automatically-generated query. Automatically-generated queries use wildcards rather than explicitly listing all of the fields in the table. This can result in slightly slower performance on the server. The wildcard (*) in automatically-generated queries is more robust to changes in the fields on the server.

Representing a table using TSQLDataSet

To make *TSQLDataSet* generate a query to fetch all fields and all records of a single database table, set the CommandType property to *ctTable*.

When *CommandType* is *ctTable*, *TSQLDataSet* generates a query based on the values of two properties:

- CommandText specifies the name of the database table that the *TSQLDataSet* object should represent.
- SortFieldNames lists the names of any fields to use to sort the data, in the order of significance.

For example, if you specify the following:

```
SQLDataSet1.CommandType := ctTable;
SQLDataSet1.CommandText := 'Employee';
SQLDataSet1.SortFieldNames := 'HireDate,Salary'
SQLDataSet1->CommandType = ctTable;
SQLDataSet1->CommandText = "Employee";
SQLDataSet1->SortFieldNames = "HireDate,Salary"
```

TSQLDataSet generates the following query, which lists all the records in the Employee table, sorted by HireDate and, within HireDate, by Salary:

```
select * from Employee order by HireDate, Salary
```

Representing a table using TSQLTable

When using *TSQLTable*, specify the table you want using the TableName property.

To specify the order of fields in the dataset, you must specify an index. There are two ways to do this:

- Set the IndexName property to the name of an index defined on the server that imposes the order you want.
- Set the IndexFieldNames property to a semicolon-delimited list of field names on which to sort. *IndexFieldNames* works like the *SortFieldNames* property of *TSQLDataSet*, except that it uses a semicolon instead of a comma as a delimiter.

See Also

Representing the Results of a Stored Procedure (see page 1833)

Representing the Results of a Query (see page 1832)

3.2.3.12.11 Representing the Results of a Query

Using a query is the most general way to specify a set of records. Queries are simply commands written in SQL. You can use either *TSQLDataSet* or *TSQLQuery* to represent the result of a query.

When using *TSQLDataSet*, set the CommandType property to *ctQuery* and assign the text of the query statement to the

CommandText property. When using *TSQLQuery*, assign the query to the SQL property instead. These properties work the same way for all general-purpose or query-type datasets. Specifying the query (see page 1613) discusses them in greater detail.

When you specify the query, it can include parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values that appear in the SQL statement. Using parameters in queries and supplying values for those parameters is discussed in Using parameters in queries (see page 1617).

SQL defines queries such as UPDATE queries that perform actions on the server but do not return records. Such queries are discussed in Executing commands that do not return records (see page 1829).

See Also

Representing the Results of a Stored Procedure (see page 1833)

3.2.3.12.12 Representing the Results of a Stored Procedure

Stored procedures are sets of SQL statements that are named and stored on an SQL server. How you indicate the stored procedure you want to execute depends on the type of unidirectional dataset you are using.

When using TSQLDataSet, to specify a stored procedure:

- Set the CommandType property to *ctStoredProc*.
- Specify the name of the stored procedure as the value of the CommandText property:

```
SQLDataSet1.CommandType := ctStoredProc;
SQLDataSet1.CommandText := 'MyStoredProcName';
SQLDataSet1->CommandType = ctStoredProc;
SQLDataSet1->CommandText = "MyStoredProcName";
```

When using TSQLStoredProc, you need only specify the name of the stored procedure as the value of the StoredProcName property.

```
SQLStoredProc1.StoredProcName := 'MyStoredProcName';
SQLStoredProc1->StoredProcName = "MyStoredProcName";
```

After you have identified a stored procedure, your application may need to enter values for any input parameters of the stored procedure or retrieve the values of output parameters after you execute the stored procedure. See Working with stored procedure parameters (see page 1625) for information about working with stored procedure parameters.

See Also

Representing the Results of a Query (see page 1832)

3.2.3.12.13 Setting Up Master/Detail Linked Relationships

There are two ways to use linked datasets to set up a master/detail relationship with a dbExpress dataset as the detail set. Which method you use depends on the type of unidirectional dataset (see page 1836) you are using. Once you have set up such a relationship, the unidirectional dataset (the "many" in a one-to-many relationship) provides access only to those records that correspond to the current record on the master set (the "one" in the one-to-many relationship).

TSQLDataSet and *TSQLQuery* require you to use a parameterized query to establish a master/detail relationship. This is the technique for creating such relationships on all query-type datasets. For details on creating master/detail relationships with query-type datasets, see Establishing master/detail relationships using parameters (see page 1593).

To set up a master/detail relationship where the detail set is an instance of *TSQLTable*, use the MasterSource and MasterFields properties, just as you would with any other table-type dataset. For details on creating master/detail relationships with table-type datasets, see Creating Master/detail Relationships (see page 1590).

See Also

Connecting to the Database Server (see page 1825)

Types of dbExpress Datasets (see page 1836)

Specifying What Data to Display (see page 1836)

Executing Commands That Do Not Return Records (see page 1829)

Accessing Schema Information (see page 51)

Fetching the Data (see page 1830)

Debugging dbExpress Applications (see page 55)

3.2.3.12.14 Setting Up TSQLConnection

In order to describe a database connection in sufficient detail for TSQLConnection to open a connection, you must identify both the driver to use and a set of connection parameters the are passed to that driver.

Identifying the driver

The driver is identified by the DriverName property, which is the name of an installed *dbExpress* driver, such as ASA, ASE, INTERBASE, INFORMIX, ORACLE, MYSQL, MSSQL, DB2 or BlackfishSQL. BlackfishSQL uses the DbxClient driver, which is written entirely in Object Pascal. The rest of these databases use Dynalink drivers, which are partially written in Object Pascal. For Dynalink drivers, the driver name is associated with these files:

- The *dbExpress* driver. This is a dynamic-link library with a name like dbx*.dll. For example, the Interbase driver DLL begins with “dbxint” and the Oracle driver DLL begins with “dbxora”
- The dynamic-link library provided by the database vendor for client-side support.

The relationship between these files and the database name is stored in a file called dbxdrivers.ini, which is updated when you install a *dbExpress* driver. Typically, you do not need to worry about these files because the SQL connection component looks them up in dbxdrivers.ini when given the value of *DriverName*. When you set the *DriverName* property, *TSQLConnection* automatically sets the LibraryName and VendorLib properties to the names of the associated dlls. Once *LibraryName* and *VendorLib* have been set, your application does not need to rely on dbxdrivers.ini. (That is, you do not need to deploy dbxdrivers.ini with your application unless you set the *DriverName* property at runtime.)

Specifying connection parameters

The Params property is a string list that lists name/value pairs. Each pair has the form *Name*=*Value*, where *Name* is the name of the parameter, and *Value* is the value you want to assign.

The particular parameters you need depend on the database server you are using. However, one particular parameter, *Database*, is required for all servers. Its value depends on the server you are using. For example, with InterBase, *Database* is the name of the .gdb file, with ORACLE it is the entry in TNSNames.ora, while with DB2, it is the client-side node name.

Other typical parameters include the *User_Name* (the name to use when logging in), *Password* (the password for *User_Name*), *HostName* (the machine name or IP address of where the server is located), and *TransIsolation* (the degree to which transactions you introduce are aware of changes made by other transactions). When you specify a driver name, the *Params* property is preloaded with all the parameters you need for that driver type, initialized to default values.

Because *Params* is a string list, at design time you can double-click on the *Params* property in the **Object Inspector** to edit the parameters using the String List editor. At runtime, use the Params.Values property to assign values to individual parameters.

Naming a connection description

Although you can always specify a connection using only the *DatabaseName* and *Params* properties, it can be more convenient to name a specific combination and then just identify the connection by name. You can name *dbExpress* database and parameter combinations, which are then saved in a file called dbxconnections.ini. The name of each combination is called a

connection name.

Once you have defined the connection name, you can identify a database connection by simply setting the ConnectionName property to a valid connection name. Setting *ConnectionName* automatically sets the *DriverName* and *Params* properties. Once *ConnectionName* is set, you can edit the *Params* property to create temporary differences from the saved set of parameter values, but changing the *DriverName* property clears both *Params* and *ConnectionName*.

One advantage of using connection names arises when you develop your application using one database (for example Local InterBase), but deploy it for use with another (such as ORACLE). In that case, *DriverName* and *Params* will likely differ on the system where you deploy your application from the values you use during development. You can switch between the two connection descriptions easily by using two versions of the dbxconnections.ini file. At design-time, your application loads the *DriverName* and *Params* from the design-time version of dbxconnections.ini. Then, when you deploy your application, it loads these values from a separate version of dbxconnections.ini that uses the "real" database. However, for this to work, you must instruct your connection component to reload the *DriverName* and *Params* properties at runtime. There are two ways to do this:

- Set the LoadParamsOnConnect property to *True*. This causes *TSQLConnection* to automatically set *DriverName* and *Params* to the values associated with *ConnectionName* in dbxconnections.ini when the connection is opened.
- Call the LoadParamsFromIniFile method. This method sets *DriverName* and *Params* to the values associated with *ConnectionName* in dbxconnections.ini (or in another file that you specify). You might choose to use this method if you want to then override certain parameter values before opening the connection.

Using the Connection Editor

The relationships between connection names and their associated driver and connection parameters is stored in the dbxconnections.ini file. You can create or modify these associations using the Connection Editor.

To display the Connection Editor, double-click on the *TSQLConnection* component. The Connection Editor appears, with a drop-down list containing all available drivers, a list of connection names for the currently selected driver, and a table listing the connection parameters for the currently selected connection name.

You can use this dialog to indicate the connection to use by selecting a driver and connection name. Once you have chosen the configuration you want, click the Test Connection button to check that you have chosen a valid configuration.

In addition, you can use this dialog to edit the named connections in dbxconnections.ini:

- Edit the parameter values in the parameter table to change the currently selected named connection. When you exit the dialog by clicking OK, the new parameter values are saved to dbxconnections.ini.
- Click the Add Connection button to define a new named connection. A dialog appears where you specify the driver to use and the name of the new connection. Once the connection is named, edit the parameters to specify the connection you want and click the OK button to save the new connection to dbxconnections.ini.
- Click the Delete Connection button to delete the currently selected named connection from dbxconnections.ini.
- Click the Rename Connection button to change the name of the currently selected named connection. Note that any edits you have made to the parameters are saved with the new name when you click the OK button.

See Also

Connecting to Databases (see page 1506)

3.2.3.12.15 Specifying the Command to Execute

With unidirectional datasets, the way you specify the command to execute is the same whether the command results in a dataset or not. That is:

When using TSQLDataSet, use the CommandType and CommandText properties to specify the command:

- If *CommandType* is *ctQuery*, *CommandText* is the SQL statement to pass to the server.
- If *CommandType* is *ctStoredProc*, *CommandText* is the name of a stored procedure to execute.

When using TSQLQuery, use the SQL property to specify the SQL statement to pass to the server.

When using TSQLStoredProc, use the StoredProcName property to specify the name of the stored procedure to execute.

Just as you specify the command in the same way as when you are retrieving records, you work with query parameters (see page 1617) or stored procedure parameters (see page 1625) the same way as with queries and stored procedures that return records.

See Also

Executing the Command (see page 1829)

Creating and Modifying Server Metadata (see page 1826)

3.2.3.12.16 Specifying What Data to Display

There are a number of ways to specify what data a dbExpress dataset represents. Which method you choose depends on the type of unidirectional dataset (see page 1836) you are using and whether the information comes from a single database table, the results of a query, or from a stored procedure.

When you work with a TSQLDataSet component, use the CommandType property to indicate where the dataset gets its data. *CommandType* can take any of the following values:

- *ctQuery*: When *CommandType* is *ctQuery*, *TSQLDataSet* executes a query you specify. If the query is a SELECT command, the dataset contains the resulting set of records.
- *ctTable*: When *CommandType* is *ctTable*, *TSQLDataSet* retrieves all of the records from a specified table.
- *ctStoredProc*: When *CommandType* is *ctStoredProc*, *TSQLDataSet* executes a stored procedure. If the stored procedure returns a cursor, the dataset contains the returned records.

The following topics describe how you can specify a set of records for each type of source:

- Representing the results of a query (see page 1832)
- Representing the records in a table (see page 1832)
- Representing the results of a stored procedure (see page 1833)

  Note: You can also populate the unidirectional dataset with metadata about what is available on the server. For information on how to do this, see Fetching metadata into a dbExpress dataset (see page 1830).

See Also

Connecting to the Database Server (see page 1825)

Types of dbExpress Datasets (see page 1836)

Fetching the Data (see page 1830)

Executing Commands That Do Not Return Records (see page 1829)

Accessing Schema Information (see page 51)

Debugging dbExpress Applications (see page 55)

3.2.3.12.17 Types of dbExpress Datasets

The *dbExpress* category of the **Tool palette** contains four types of unidirectional dataset: TSQLDataSet, TSQLQuery, TSQLTable, and TSQLStoredProc.

TSQLDataSet is the most general of the four. You can use an SQL dataset to represent any data available through *dbExpress*, or to send commands to a database accessed through *dbExpress*. This is the recommended component to use for working with database tables in new database applications.

TSQLQuery is a query-type dataset (see page 1618) that encapsulates an SQL statement and enables applications to access the resulting records, if any.

TSQLTable is a table-type dataset (see page 1620) that represents all of the rows and columns of a single database table.

TSQLStoredProc is a stored procedure-type dataset (see page 1618) that executes a stored procedure defined on a database server.

Note: The *dbExpress* page also includes TSimpleDataSet, which is not a unidirectional dataset. Rather, it is a client dataset that uses a unidirectional dataset internally to access its data.

See Also

Specifying What Data to Display (see page 1836)

Connecting to the Database Server (see page 1825)

Fetching the Data (see page 1830)

Executing Commands That Do Not Return Records (see page 1829)

Accessing Schema Information (see page 51)

Debugging dbExpress Applications (see page 55)

3.2.3.12.18 The Structure of Metadata Datasets

For each type of metadata you can access using *TSQLDataSet*, there is a predefined set of columns (fields) that are populated with information about the items of the requested type.

Information about tables

When you request information about tables (*stTables* or *stSysTables*), the resulting dataset includes a record for each table. It has the following columns:

Columns in tables of metadata listing tables

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the table.
TABLE_NAME	ftString	The name of the table. This field determines the sort order of the dataset.
TABLE_TYPE	ftInteger	Identifies the type of table. It is a sum of one or more of the following values: 1: Table 2: View 4: System table 8: Synonym 16: Temporary table 32: Local table.

Information about stored procedures

When you request information about stored procedures (*stProcedures*), the resulting dataset includes a record for each stored procedure. It has following columns:

Columns in tables of metadata listing stored procedures

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.

CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure. This field determines the sort order of the dataset.
PROC_TYPE	ftInteger	Identifies the type of stored procedure. It is a sum of one or more of the following values: 1: Procedure 2: Function 4: Package 8: System procedure
IN_PARAMS	ftSmallint	The number of input parameters
OUT_PARAMS	ftSmallint	The number of output parameters.

Information about fields

When you request information about the fields in a specified table (*stColumns*), the resulting dataset includes a record for each field. It includes the following columns:

Columns in tables of metadata listing fields

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the table whose fields you listing. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the field.
TABLE_NAME	ftString	The name of the table that contains the fields.
COLUMN_NAME	ftString	The name of the field. This value determines the sort order of the dataset.
COLUMN_POSITION	ftSmallint	The position of the column in its table.
COLUMN_TYPE	ftInteger	Identifies the type of value in the field. It is a sum of one or more of the following: 1: Row ID 2: Row Version 4: Auto increment field 8: Field with a default value
COLUMN_DATATYPE	ftSmallint	The datatype of the column. This is one of the logical field type constants defined in sqllinks.pas.
COLUMN_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in COLUMN_DATATYPE and COLUMN_SUBTYPE, but in a form used in some DDL statements.
COLUMN_SUBTYPE	ftSmallint	A subtype for the column's datatype. This is one of the logical subtype constants defined in sqllinks.pas.
COLUMN_PRECISION	ftInteger	The size of the field type (number of characters in a string, bytes in a bytes field, significant digits in a BCD value, members of an ADT field, and so on).
COLUMN_SCALE	ftSmallint	The number of digits to the right of the decimal on BCD values, or descendants on ADT and array fields.
COLUMN_LENGTH	ftInteger	The number of bytes required to store field values.
COLUMN_NULLABLE	ftSmallint	A Boolean that indicates whether the field can be left blank (0 means the field requires a value).

Information about indexes

When you request information about the indexes on a table (stIndexes), the resulting dataset includes a record for each field in each record. (Multi-record indexes are described using multiple records) The dataset has the following columns:

Columns in tables of metadata listing indexes

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the index. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the index.
TABLE_NAME	ftString	The name of the table for which the index is defined.
INDEX_NAME	ftString	The name of the index. This field determines the sort order of the dataset.
PKEY_NAME	ftString	Indicates the name of the primary key.
COLUMN_NAME	ftString	The name of the field (column) in the index.
COLUMN_POSITION	ftSmallint	The position of this field in the index.
INDEX_TYPE	ftSmallint	Identifies the type of index. It is a sum of one or more of the following values: 1: Non-unique 2: Unique 4: Primary key
SORT_ORDER	ftString	Indicates that the index is ascending (a) or descending (d).
FILTER	ftString	Describes a filter condition that limits the indexed records.

Information about stored procedure parameters

When you request information about the parameters of a stored procedure (*stProcedureParams*), the resulting dataset includes a record for each parameter. It has the following columns:

Columns in tables of metadata listing parameters

Column name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the stored procedure. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the stored procedure.
PROC_NAME	ftString	The name of the stored procedure that contains the parameter.
PARAM_NAME	ftString	The name of the parameter. This field determines the sort order of the dataset.
PARAM_TYPE	ftSmallint	Identifies the type of parameter. This is the same as a *TParam* object's *ParamType* property.
PARAM_DATATYPE	ftSmallint	The datatype of the parameter. This is one of the logical field type constants defined in sqllinks.pas.
PARAM_SUBTYPE	ftSmallint	A subtype for the parameter's datatype. This is one of the logical subtype constants defined in sqllinks.pas.
PARAM_TYPENAME	ftString	A string describing the datatype. This is the same information as contained in PARAM_DATATYPE and PARAM_SUBTYPE, but in a form used in some DDL statements.
PARAM_PRECISION	ftInteger	The maximum number of digits in floating-point values or bytes (for strings and Bytes fields).
PARAM_SCALE	ftSmallint	The number of digits to the right of the decimal on floating-point values.
PARAM_LENGTH	ftInteger	The number of bytes required to store parameter values.
PARAM_NULLABLE	ftSmallint	A Boolean that indicates whether the parameter can be left blank (0 means the parameter requires a value).

Information about Oracle packages

Columns in tables of metadata listing stored procedures

Column Name	Field type	Contents
RECNO	ftInteger	A record number that uniquely identifies each record.
CATALOG_NAME	ftString	The name of the catalog (database) that contains the package. This is the same as the *Database* parameter on an SQL connection component.
SCHEMA_NAME	ftString	The name of the schema that identifies the owner of the package.
OBJECT_NAME	ftString	The name of the package. This field determines the sort order of the dataset.

See Also

Accessing Schema Information (see page 1824)

3.2.3.13 Using XML in database applications

Topics

Name	Description
Converting XML Documents into Data Packets (see page 1841)	Once you have created a transformation file (see page 1843) that indicates how to transform an XML document into a data packet, you can create data packets for any XML document that conforms to the schema used in the transformation. These data packets can then be assigned to a client dataset and saved to a file so that they form the basis of a file-based database application (see page 1564). The TXMLTransform component transforms an XML document into a data packet according to the mapping in a transformation file. **Note:** You can also use *TXMLTransform* to convert a data packet that appears in XML format... more (see page 1841)
Defining Transformations (see page 1843)	Before you can convert between data packets and XML documents, you must define the relationship between the metadata in a data packet and the nodes of the corresponding XML document (see page 1843). A description of this relationship is stored in a special XML document called a transformation. Each transformation file contains two things: the mapping between the nodes in an XML schema and the fields in a data packet, and a skeletal XML document that represents the structure for the results of the transformation. A transformation is a one-way mapping: from an XML schema or document to a data packet or... more (see page 1843)
Mapping Between XML Nodes and Data Packet Fields (see page 1843)	XML provides a text-based way to store or describe structured data. Datasets provide another way to store and describe structured data. To convert an XML document into a dataset, therefore, you must identify the correspondences between the nodes in an XML document and the fields in a dataset. Consider, for example, an XML document that represents a set of email messages. It might look like the following (containing a single message):
Using an XML Document as the Client of a Provider (see page 1845)	The TXMLTransformClient component acts as an adapter to let you use an XML document (or set of documents) as the client for an application server (or simply as the client of a dataset to which it connects via a *TDataSetProvider* component). That is, *TXMLTransformClient* lets you publish database data as an XML document and to make use of update requests (insertions or deletions) from an external application that supplies them in the form of XML documents. To specify the provider from which the *TXMLTransformClient* object fetches data and to which it applies updates, set the ProviderName property. As with the... more (see page 1845)
Using an XML Document as the Source for a Provider (see page 1846)	The TXMLTransformProvider component lets you use an XML document as if it were a database table. *TXMLTransformProvider* packages the data from an XML document and applies updates from clients back to that XML document. It appears to clients such as client datasets or XML brokers like any other provider component. For information on provider components, see Using Provider Components (see page 1819). For information on using provider components with client datasets, see Using a Client Dataset with a Provider (see page 1732). You can specify the XML document from which the XML provider provides data and to which it applies updates using the XMLDataFile... more (see page 1846)

Using XML in Database Applications (see page 1847)	In addition to the support for connecting to database servers (see page 1506), Delphi lets you work with XML documents as if they were database servers. XML (Extensible Markup Language) is a markup language for describing structured data. XML documents provide a standard, transportable format for data that is used in Web applications, business-to-business communication, and so on. For information on Delphi's support for working directly with XML documents, see Working with XML Documents. (see page 2352) Support for working with XML documents in database applications is based on a set of components that can convert data packets (the *Data* property of a client dataset)... more (see page 1847)
Using XMLMapper (see page 1848)	The XML mapper utility, xmlmapper.exe, lets you define mappings in three ways: • From an existing XML schema (or document) to a client dataset that you define. This is useful when you want to create a database application to work with data for which you already have an XML schema. • From an existing data packet to a new XML schema you define. This is useful when you want to expose existing database information in XML, for example to create a new business-to-business communication system. • Between an existing XML schema and an existing data packet. This is useful when you have an... more (see page 1848)

3.2.3.13.1 Converting XML Documents into Data Packets

Once you have created a transformation file (see page 1843) that indicates how to transform an XML document into a data packet, you can create data packets for any XML document that conforms to the schema used in the transformation. These data packets can then be assigned to a client dataset and saved to a file so that they form the basis of a file-based database application (see page 1564).

The TXMLTransform component transforms an XML document into a data packet according to the mapping in a transformation file.

Note: You can also use *TXMLTransform* to convert a data packet that appears in XML format into an arbitrary XML document.

Specifying the source XML document

There are three ways to specify the source XML document:

- If the source document is an .xml file on disk, you can use the SourceXmlFile property.
- If the source document is an in-memory string of XML, you can use the SourceXml property.
- If you have an IDOMDocument interface for the source document, you can use the SourceXmlDocument property.

TXMLTransform checks these properties in the order listed above. That is, it first checks for a file name in the *SourceXmlFile* property. Only if *SourceXmlFile* is an empty string does it check the *SourceXml* property. Only if *SourceXml* is an empty string does it then check the *SourceXmlDocument* property.

Specifying the transformation

There are two ways to specify the transformation that converts the XML document into a data packet:

- Set the TransformationFile property to indicate a transformation file that was created using xmlmapper.exe.
- Set the TransformationDocument property if you have an *IDOMDocument* interface for the transformation.

TXMLTransform checks these properties in the order listed above. That is, it first checks for a file name in the *TransformationFile* property. Only if *TransformationFile* is an empty string does it check the *TransformationDocument* property.

Obtaining the resulting data packet

To cause *TXMLTransform* to perform its transformation and generate a data packet, you need only read the Data property. For example, the following code uses an XML document and transformation file to generate a data packet, which is then assigned to

a client dataset:

```
XMLTransform1.SourceXMLFile := 'CustomerDocument.xml';
XMLTransform1.TransformationFile := 'CustXMLToCustTable.xtr';
ClientDataSet1.XMLData := XMLTransform1.Data;
XMLTransform1->SourceXMLFile = "CustomerDocument.xml";
XMLTransform1->TransformationFile = "CustXMLToCustTable.xtr";
ClientDataSet1->XMLData = XMLTransform1->Data;
```

Converting user-defined nodes

When you define a transformation using xmlmapper.exe (see page 1848), you can specify that some of the nodes in the XML document are "user-defined." User-defined nodes are nodes for which you want to provide the transformation in code rather than relying on a straightforward node-value-to-field-value translation.

You can provide the code to translate user-defined nodes using the OnTranslate event. The *OnTranslate* event handler is called every time the *TXMLTransform* component encounters a user-defined node in the XML document. In the OnTranslate event handler, you can read the source document and specify the resulting value for the field in the data packet.

For example, the following *OnTranslate* event handler converts a node in the XML document with the following form

```
<FullName>
   <Title> </Title>
   <FirstName> </FirstName>
   <LastName> </LastName>
</FullName>
```

into a single field value:

```
procedure TForm1.XMLTransform1Translate(Sender: TObject; Id: String; SrcNode: IDOMNode;
  var Value: String; DestNode: IDOMNode);
var
  CurNode: IDOMNode;
begin
  if Id = 'FullName' then
  begin
    Value = '';
    if SrcNode.hasChildNodes then
    begin
      CurNode := SrcNode.firstChild;
      Value := Value + CurNode.nodeValue;
      while CurNode <> SrcNode.lastChild do
      begin
        CurNode := CurNode.nextSibling;
        Value := Value + ' ';
        Value := Value + CurNode.nodeValue;
      end;
    end;
  end;
end;
void __fastcall TForm1::XMLTransform1Translate(TObject *Sender, AnsiString Id,
_di_IDOMNode SrcNode, AnsiString &Value, _di_IDOMNode DestNode)
{
  if (Id == "FullName")
  {
    Value = "";
    if (SrcNode.hasChildNodes)
    {
      _di_IXMLDOMNode CurNode = SrcNode.firstChild;
      Value = SrcValue + AnsiString(CurNode.nodeValue);
      while (CurNode != SrcNode.lastChild)
      {
        CurNode = CurNode.nextSibling;
        Value = Value + AnsiString(" ");
        Value = Value + AnsiString(CurNode.nodeValue);
      }
    }
```

```
  }
}
```

See Also

Defining Transformations (see page 1843)

Using an XML Document as the Source for a Provider (see page 1846)

Using an XML Document as the Client of a Provider (see page 1845)

3.2.3.13.2 Defining Transformations

Before you can convert between data packets and XML documents, you must define the relationship between the metadata in a data packet and the nodes of the corresponding XML document (see page 1843). A description of this relationship is stored in a special XML document called a transformation.

Each transformation file contains two things: the mapping between the nodes in an XML schema and the fields in a data packet, and a skeletal XML document that represents the structure for the results of the transformation. A transformation is a one-way mapping: from an XML schema or document to a data packet or from the metadata in a data packet to an XML schema. Often, you create transformation files in pairs: one that maps from XML to data packet, and one that maps from data packet to XML.

In order to create the transformation files for a mapping, use the XMLMapper utility (see page 1848) that ships in the bin directory.

See Also

Converting XML Documents into Data Packets (see page 1841)

Using an XML Document as the Source for a Provider (see page 1846)

Using an XML Document as the Client of a Provider (see page 1845)

3.2.3.13.3 Mapping Between XML Nodes and Data Packet Fields

XML provides a text-based way to store or describe structured data. Datasets provide another way to store and describe structured data. To convert an XML document into a dataset, therefore, you must identify the correspondences between the nodes in an XML document and the fields in a dataset.

Consider, for example, an XML document that represents a set of email messages. It might look like the following (containing a single message):

```
<?xml version="1.0" standalone="yes" ?>
<email>
   <head>
      <from>
         <name>Dave Boss</name>
         <address>dboss@MyCo.com</address>
      </from>
      <to>
         <name>Joe Engineer</name>
         <address>jengineer@MyCo.com</address>
      </to>
      <cc>
         <name>Robin Smith/name>
         <address>rsmith@MyCo.com</address>
      </cc>
      <cc>
         <name>Leonard Devon</name>
         <address>ldevon@MyCo.com</address>
      </cc>
   </head>
```

```
   <body>
      <subject>XML components</subject>
      <content>
        Joe,
        Attached is the specification for the XML component support in Delphi.
        This looks like a good solution to our buisness-to-buisness application!
        Also attached, please find the project schedule. Do you think its reasonable?
           Dave.
      </content>
      <attachment attachfile="XMLSpec.txt"/>
      <attachment attachfile="Schedule.txt"/>
   </body>
</email>
```

One natural mapping between this document and a dataset would map each e-mail message to a single record. The record would have fields for the sender's name and address. Because an e-mail message can have multiple recipients, the recipient (<to> would map to a nested dataset. Similarly, the cc list maps to a nested dataset. The subject line would map to a string field while the message itself (<content>) would probably be a memo field. The names of attachment files would map to a nested dataset because one message can have several attachments. Thus, the e-mail above would map to a dataset something like the following:

SenderName	SenderAddress	To	CC	Subject	Content	Attach
Dave Boss	dboss@MyCo.Com	(DataSet)	(DataSet)	XML components	(MEMO)	(DataSet)

where the nested dataset in the "To" field is

Name	Address
Joe Engineer	jengineer@MyCo.Com

the nested dataset in the "CC" field is

Name	Address
Robin Smith	rsmith@MyCo.Com
Leonard Devon	ldevon@MyCo.Com

and the nested dataset in the "Attach" field is

Attachfile
XMLSpec.txt
Schedule.txt

Defining such a mapping involves identifying those nodes of the XML document that can be repeated and mapping them to nested datasets. Tagged elements that have values and appear only once (such as <content>...</content>) map to fields whose datatype reflects the type of data that can appear as the value. Attributes of a tag (such as the AttachFile attribute of the attachment tag) also map to fields.

Note that not all tags in the XML document appear in the corresponding dataset. For example, the <head>...<head/> element has no corresponding element in the resulting dataset. Typically, only elements that have values, elements that can be repeated, or the attributes of a tag map to the fields (including nested dataset fields) of a dataset. The exception to this rule is when a parent node in the XML document maps to a field whose value is built up from the values of the child nodes. For example, an XML document might contain a set of tags such as

```
<FullName>
   <Title> Mr. </Title>
   <FirstName> John </FirstName>
   <LastName> Smith </LastName>
```

```
</FullName>
```

which could map to a single dataset field with the value

```
Mr. John Smith
```

See Also

Using XMLMapper (see page 1848)

3.2.3.13.4 Using an XML Document as the Client of a Provider

The TXMLTransformClient component acts as an adapter to let you use an XML document (or set of documents) as the client for an application server (or simply as the client of a dataset to which it connects via a *TDataSetProvider* component). That is, *TXMLTransformClient* lets you publish database data as an XML document and to make use of update requests (insertions or deletions) from an external application that supplies them in the form of XML documents.

To specify the provider from which the *TXMLTransformClient* object fetches data and to which it applies updates, set the ProviderName property. As with the *ProviderName* property of a client dataset, *ProviderName* can be the name of a provider on a remote application server or it can be a local provider in the same form or data module as the *TXMLTransformClient* object. For information about providers, see Using Provider Components (see page 1819).

If the provider is on a remote application server, you must use a DataSnap connection component to connect to that application server. Specify the connection component using the RemoteServer property. For information on DataSnap connection components, see Connecting to the Application Server (see page 1517).

Fetching an XML document from a provider

TXMLTransformClient uses an internal *TXMLTransform* component to translate data packets from the provider into an XML document. You can access this *TXMLTransform* component as the value of the TransformGetData property.

Before you can create an XML document that represents the data from a provider, you must specify the transformation file (see page 1843) that *TransformGetData* uses to translate the data packet into the appropriate XML format. You do this by setting the *TXMLTransform* component's TransformationFile or TransformationDocument property, just as when using a stand-alone TXMLTransform component (see page 1841). If that transformation includes any user-defined nodes, you will want to supply *TransformGetData* with an OnTranslate event handler as well.

There is no need to specify the source document for *TransformGetData*, *TXMLTransformClient* fetches that from the provider. However, if the provider expects any input parameters, you may want to set them before fetching the data. Use the SetParams method to supply these input parameters before you fetch data from the provider. *SetParams* takes two arguments: a string of XML from which to extract parameter values, and the name of a transformation file to translate that XML into a data packet. *SetParams* uses the transformation file to convert the string of XML into a data packet, and then extracts the parameter values from that data packet.

Note: You can override either of these arguments if you want to specify the parameter document or transformation in another way. Simply set one of the properties on TransformSetParams property to indicate the document that contains the parameters or the transformation to use when converting them, and then set the argument you want to override to an empty string when you call *SetParams*. For details on the properties you can use, see Converting XML Documents Into Data Packets (see page 1841).

Once you have configured *TransformGetData* and supplied any input parameters, you can call the GetDataAsXml method to fetch the XML. *GetDataAsXml* sends the current parameter values to the provider, fetches a data packet, converts it into an XML document, and returns that document as a string. You can save this string to a file:

```
var
  XMLDoc: TFileStream;
  XML: string;
begin
  XMLTransformClient1.ProviderName := 'Provider1';
```

```
    XMLTransformClient1.TransformGetData.TransformationFile := 'CustTableToCustXML.xtr';
    XMLTransformClient1.TransFormSetParams.SourceXmlFile := 'InputParams.xml';
    XMLTransformClient1.SetParams('', 'InputParamsToDP.xtr');
    XML := XMLTransformClient1.GetDataAsXml('');
    XMLDoc := TFileStream.Create('Customers.xml', fmCreate or fmOpenWrite);
    try
      XMLDoc.Write(XML, Length(XML));
    finally
      XMLDoc.Free;
    end;
  end;
  XMLTransformClient1->ProviderName = "Provider1";
  XMLTransformClient1->TransformGetData->TransformationFile = "CustTableToCustXML.xtr";
  XMLTransformClient1->TransFormSetParams->SourceXmlFile = "InputParams.xml";
  XMLTransformClient1->SetParams("", "InputParamsToDP.xtr");
  AnsiString XML = XMLTransformClient1->GetDataAsXml("");
  TFileStream pXMLDoc = new TFileStream("Customers.xml", fmCreate || fmOpenWrite);
  __try
  {
    pXMLDoc->Write(XML.c_str(), XML.Length());
  }
  __finally
  {
    delete pXMLDoc;
  }
```

Applying updates from an XML document to a provider

TXMLTransformClient also lets you insert all of the data from an XML document into the provider's dataset or to delete all of the records in an XML document from the provider's dataset. To perform these updates, call the ApplyUpdates*method, passing in*

- A string whose value is the contents of the XML document with the data to insert or delete.
- The name of a transformation file that can convert that XML data into an insert or delete delta packet. (When you define the transformation file using the XML mapper utility, you specify whether the transformation is for an insert or delete delta packet.)
- The number of update errors that can be tolerated before the update operation is aborted. If fewer than the specified number of records can't be inserted or deleted, *ApplyUpdates* returns the number of actual failures. If more than the specified number of records can't be inserted or deleted, the entire update operation is rolled back, and no update is performed.

The following call transforms the XML document Customers.xml into a delta packet and applies all updates regardless of the number of errors:

```
StringList1.LoadFromFile('Customers.xml');
nErrors := ApplyUpdates(StringList1.Text, 'CustXMLToInsert.xtr', -1);
StringList1->LoadFromFile("Customers.xml");
nErrors = ApplyUpdates(StringList1->Text, "CustXMLToInsert.xtr", -1);
```

See Also

Defining Transformations (see page 1843)

Converting XML Documents into Data Packets (see page 1841)

Using an XML Document as the Source for a Provider (see page 1846)

3.2.3.13.5 Using an XML Document as the Source for a Provider

The TXMLTransformProvider component lets you use an XML document as if it were a database table. *TXMLTransformProvider* packages the data from an XML document and applies updates from clients back to that XML document. It appears to clients such as client datasets or XML brokers like any other provider component. For information on provider components, see Using Provider Components (see page 1819). For information on using provider components with client datasets, see Using a Client Dataset with a Provider (see page 1732).

You can specify the XML document from which the XML provider provides data and to which it applies updates using the

XMLDataFile property.

TXMLTransformProvider components use internal *TXMLTransform* components to translate between data packets and the source XML document: one to translate the XML document into data packets, and one to translate data packets back into the XML format of the source document after applying updates. These two *TXMLTransform* components can be accessed using the TransformRead and TransformWrite properties, respectively.

When using *TXMLTransformProvider*, you must specify the transformations that these two *TXMLTransform* components use to translate between data packets and the source XML document. You do this by setting the *TXMLTransform* component's TransformationFile or TransformationDocument property, just as when using a stand-alone TXMLTransform component (see page 1841).

In addition, if the transformation includes any user-defined nodes, you must supply an OnTranslate event handler to the internal *TXMLTransform* components.

You do not need to specify the source document on the *TXMLTransform* components that are the values of *TransformRead* and *TransformWrite*. For *TransformRead*, the source is the file specified by the provider's *XMLDataFile* property (although, if you set *XMLDataFile* to an empty string, you can supply the source document using *TransformRead.XmlSource* or *TransformRead.XmlSourceDocument*). For *TransformWrite*, the source is generated internally by the provider when it applies updates.

See Also

Defining Transformations (see page 1843)

Converting XML Documents into Data Packets (see page 1841)

Using an XML Document as the Client of a Provider (see page 1845)

3.2.3.13.6 Using XML in Database Applications

In addition to the support for connecting to database servers (see page 1506), Delphi lets you work with XML documents as if they were database servers. XML (Extensible Markup Language) is a markup language for describing structured data. XML documents provide a standard, transportable format for data that is used in Web applications, business-to-business communication, and so on. For information on Delphi's support for working directly with XML documents, see Working with XML Documents. (see page 2352)

Support for working with XML documents in database applications is based on a set of components that can convert data packets (the *Data* property of a client dataset) into XML documents and convert XML documents into data packets. To use these components, you must first define the transformation (see page 1843) between the XML document and the data packet. Once you have defined the transformation, you can use special components to

- convert XML documents into data packets (see page 1841).
- provide data from and resolve updates to an XML document (see page 1846).
- use an XML document as the client of a provider (see page 1845).

See Also

Designing Database Applications (see page 1566)

Working with XML Documents (see page 2352)

Understanding Datasets (see page 1632)

Using Data Controls (see page 1778)

Working with Field Components (see page 1877)

Creating and Using a Client Dataset (see page 1740)

Working With ADO Components (see page 1494)

Using Unidirectional Datasets (see page 1823)

Using the Borland Database Engine (see page 1643)

Using Provider Components (see page 1819)

Creating Multi-tiered Applications: Overview (see page 1518)

Using Decision Support Components (see page 1800)

3.2.3.13.7 Using XMLMapper

The XML mapper utility, xmlmapper.exe, lets you define mappings in three ways:

- From an existing XML schema (or document) to a client dataset that you define. This is useful when you want to create a database application to work with data for which you already have an XML schema.
- From an existing data packet to a new XML schema you define. This is useful when you want to expose existing database information in XML, for example to create a new business-to-business communication system.
- Between an existing XML schema and an existing data packet. This is useful when you have an XML schema and a database that both describe the same information and you want to make them work together.

  Note: XML mapper relies on two .DLLs (midas.dll and msxml.dll) to work correctly. Be sure that you have both of these .DLLs installed before you try to use xmlmapper.exe. In addition, msxml.dll must be registered as a COM server. You can register it using Regsvr32.exe.

Loading an XML schema or data packet

Before you can define a mapping and generate a transformation file, you must first load descriptions of the XML document and the data packet between which you are mapping.

You can load an XML document or schema by choosing **File** ▸ **Open** and selecting the document or schema in the resulting dialog.

You can load a data packet by choosing **File** ▸ **Open** and selecting a data packet file in the resulting dialog. (The data packet is simply the file generated when you call a client dataset's *SaveToFile* method.) If you have not saved the data packet to disk, you can fetch the data packet directly from the application server of a multi-tiered application by right-clicking in the Datapacket pane and choosing Connect To Remote Server.

You can load only an XML document or schema, only a data packet, or you can load both. If you load only one side of the mapping, XML mapper can generate a natural mapping for the other side.

Defining mappings

The mapping between an XML document and a data packet need not include all of the fields in the data packet or all of the tagged elements in the XML document. Therefore, you must first specify those elements that are mapped. To specify these elements, first select the Mapping page in the central pane of the dialog.

To specify the elements of an XML document or schema that are mapped to fields in a data packet, select the Sample or Structure tab of the XML document pane and double-click on the nodes for elements that map to data packet fields.

To specify the fields of the data packet that are mapped to tagged elements or attributes in the XML document, double-click on the nodes for those fields in the Datapacket pane.

If you have only loaded one side of the mapping (the XML document or the data packet), you can generate the other side after you have selected the nodes that are mapped.

- If you are generating a data packet from an XML document, you first define attributes for the selected nodes that determine the types of fields to which they correspond in the data packet. In the center pane, select the Node Repository page. Select each node that participates in the mapping and indicate the attributes of the corresponding field. If the mapping is not

straightforward (for example, a node with subnodes that corresponds to a field whose value is built from those subnodes), check the User Defined Translation check box. You will need to write an event handler later to perform the transformation on user defined nodes. Once you have specified the way nodes are to be mapped, choose **Create ▸ Datapacket from XML**. The corresponding data packet is automatically generated and displayed in the Datapacket pane.

- If you are generating an XML document from a data packet, choose **Create ▸ XML from Datapacket**. A dialog appears where you can specify the names of the tags and attributes in the XML document that correspond to fields, records, and datasets in the data packet. For field values, the way you name them indicates whether they map to a tagged element with a value or to an attribute. Names that begin with an @ symbol map to attributes of the tag that corresponds to the record, while names that do not begin with an @ symbol map to tagged elements that have values and that are nested within the element for the record.
- If you have loaded both an XML document and a data packet (client dataset file), be sure you select corresponding nodes in the same order. The corresponding nodes should appear next to each other in the table at the top of the Mapping page.

Once you have loaded or generated both the XML document and the data packet and selected the nodes that appear in the mapping, the table at the top of the Mapping page should reflect the mapping you have defined.

Generating transformation files

Once you define the mapping, you can generate the transformation files that are used to convert XML documents to data packets and to convert data packets to XML documents. Note that only the transformation file is directional: a single mapping can be used to generate both the transformation from XML to data packet and from data packet to XML.

To generate a transformation file

1. First select the radio button that indicates what the transformation creates:
   - Choose the Datapacket to XML button if the mapping goes from data packet to XML document.
   - Choose the XML to Datapacket button if the mapping goes from XML document to data packet.
2. If you are generating a data packet, you will also want to use the radio buttons in the Create Datapacket As section. These buttons let you specify how the data packet will be used: as a dataset, as a delta packet for applying updates, or as the parameters to supply to a provider before fetching data.
3. Click Create and Test Transformation to generate an in-memory version of the transformation. XML mapper displays the XML document that would be generated for the data packet in the Datapacket pane or the data packet that would be generated for the XML document in the XML Document pane.
4. Finally, choose **File ▸ Save ▸ Transformation** to save the transformation file. The transformation file is a special XML file (with the .xtr extension) that describes the transformation you have defined.

See Also

Mapping Between XML Nodes and Data Packet Fields (see page 1843)

3.2.3.14 Working with field components

Topics

Name	Description
Accessing Field Values with a Dataset's FieldByName Method (see page 1853)	You can access the value of a field with a dataset's FieldByName method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time. To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion property, such as *AsString* or *AsInteger*. For example, the following statement assigns the... more (see page 1853)

Accessing Field Values with a Dataset's Fields Property (see page 1854)	You can access the value of a field with the Fields property of the dataset component to which the field belongs. *Fields* maintains an indexed list of all the fields in the dataset. Accessing field values with the *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time. To use the *Fields* property you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first... more (see page 1854)
Arranging Persistent Fields (see page 1854)	The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.
Associating Attribute Sets with Field Components (see page 1854)	When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), and you have saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.
Creating a Custom Constraint (see page 1855)	A custom constraint is not imported from the server like other constraints. It is a constraint that you declare, implement, and enforce in your local application. As such, custom constraints can be useful for offering a prevalidation enforcement of data entry, but a custom constraint cannot be applied against data received from or sent to a server application. To create a custom constraint, set the *CustomConstraint* property to specify a constraint condition, and set *ConstraintErrorMessage* to the message to display when a user violates the constraint at runtime. *CustomConstraint* is an SQL string that specifies any application-specific constraints imposed on... more (see page 1855)
Defining a Data Field (see page 1855)	A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a TSmallIntField with a TIntegerField. Because you cannot change a field's data type directly, you must define a new field to replace it. **Warning:** Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.
Defining an Aggregate Field (see page 1856)	An aggregate field displays values from a maintained aggregate in a client dataset. An aggregate is a calculation that summarizes the data in a set of records. See Using maintained aggregates (see page 1735) for details about maintained aggregates.
Defining New Persistent Fields (see page 1856)	Besides making existing dataset fields into persistent fields, you can also create special persistent fields as additions to or replacements of the other persistent fields in a dataset. New persistent fields that you create are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in the database, or because they are temporary. The physical structure of the data underlying the dataset is not changed in any way. To create a new persistent field component, invoke the context menu for the Fields editor by right clicking and choose New field. The... more (see page 1856)
Deleting Persistent Field Components (see page 1857)	Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table.
Displaying, Converting, and Accessing Field Values (see page 1858)	Data-aware controls such as TDBEdit and TDBGrid automatically display the values associated with field components. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications. For more information about data-aware control, see Using data controls (see page 1778). Standard controls can also display and edit database values associated with field components. Using standard controls,... more (see page 1858)
Displaying Field Component Values in Standard Controls (see page 1858)	An application can access the value of a dataset column through the *Value* property of a field component. For example, the following *OnDataChange* event handler updates the text in a TEdit control because the value of the *CustomersCompany* field may have changed:

Persistent Field Components (see page 1859)	By default, dataset fields are dynamic. Their properties and availability are automatically set and cannot be changed in any way. To gain control over a field's properties and events you must create persistent fields for the dataset. Persistent fields let you • Set or change the field's display or edit characteristics at design time or runtime. • Create new fields, such as lookup fields, calculated fields, and aggregated fields, that base their values on existing fields in a dataset. • Validate data entry. • Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying... more (see page 1859)
Removing Attribute Associations (see page 1860)	If you change your mind about associating an attribute set with a field, you can remove the association.
Setting a Default Value for a Field (see page 1860)	You can specify how a default value for a field in a client dataset or a BDE-enabled dataset should be calculated at runtime using the DefaultExpression property. *DefaultExpression* can be any valid SQL value expression that does not refer to field values. If the expression contains literals other than numeric values, they must appear in quotes. For example, a default value of noon for a time field would be
Setting Display and Edit Properties at Design Time (see page 1860)	To edit the display properties of a selected field component, switch to the Properties page on the **Object Inspector** window. The following table summarizes display properties that can be edited. ***Field component properties***
Using Object Fields (see page 1862)	Object fields are fields that represent a composite of other, simpler datatypes. These include ADT (Abstract Data Type) fields, Array fields, DataSet fields, and Reference fields. All of these field types either contain or reference child fields or other data sets. ADT fields and array fields are fields that contain child fields. The child fields of an ADT field can be any scalar or object type (that is, any other field type). These child fields may differ in type from each other. An array field contains an array of child fields, all of the same type. Dataset and reference fields... more (see page 1862)
Using Server Constraints (see page 1863)	Most production SQL databases use constraints to impose conditions on the possible values for a field. For example, a field may not permit NULL values, may require that its value be unique for that column, or that its values be greater than *0* and less than *150*. While you could replicate such conditions in your client applications, client datasets and BDE-enabled datasets offer the *ImportedConstraint* property to propagate a server's constraints locally. *ImportedConstraint* is a read-only property that specifies an SQL clause that limits field values in some manner. For example:
Working with Constraints (see page 1863)	Field components in client datasets or BDE-enabled datasets can use SQL server constraints (see page 1863). In addition, your applications can create and use custom constraints (see page 1855) for these datasets that are local to your application. All constraints are rules or conditions that impose a limit on the scope or range of values that a field can store.
Working with Field Component Methods at Runtime (see page 1864)	Field components methods available at runtime enable you to convert field values from one data type to another, and enable you to set focus to the first data-aware control in a form that is associated with a field component. Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as BeforePost). Validation may be performed on the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control... more (see page 1864)
Working with ADT Fields (see page 1864)	ADTs are user-defined types created on the server, and are similar to the record type. An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs. There are a variety of ways to access the data in ADT field types. These are illustrated in the following examples, which assign a child field value to an edit box called CityEdit, and use the following ADT structure,

Working with Array Fields (see page 1866)	Array fields consist of a set of fields of the same type. The field types can be scalar (for example, float, string), or non-scalar (an ADT), but an array field of arrays is not permitted. The SparseArrays property of *TDataSet* determines whether a unique TField object is created for each element of the array field. There are a variety of ways to access the data in array field types. If you are not using persistent fields, the dataset's ObjectView property must be set to True before you can access the elements of an array field.
Working with DataSet Fields (see page 1867)	Dataset fields provide access to data stored in a nested dataset. The NestedDataSet property references the nested dataset. The data in the nested dataset is then accessed through the field objects of the nested dataset.
Working with Reference Fields (see page 1867)	Reference fields store a pointer or reference to another ADT object. This ADT object is a single record of another object table. Reference fields always refer to a single record in a dataset (object table). The data in the referenced object is actually returned in a nested dataset, but can also be accessed via the Fields property on the *TReferenceField*.
Defining a Lookup Field (see page 1868)	A lookup field is a read-only field that displays values at runtime based on search criteria you specify. In its simplest form, a lookup field is passed the name of an existing field to search on, a field value to search for, and a different field in a lookup dataset whose value it should display. For example, consider a mail-order application that enables an operator to use a lookup field to determine automatically the city and state that correspond to the zip code a customer provides. The column to search on might be called *ZipTable.Zip*, the value to search... more (see page 1868)
Creating Attribute Sets for Field Components (see page 1870)	When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields. **Note:** Attribute sets and the Data Dictionary are only available for BDE-enabled datasets.
Programming a Calculated Field (see page 1870)	After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.
Setting Persistent Field Properties and Events (see page 1871)	You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a TDBGrid, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated. To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the **Object Inspector**. The following topics discuss using persistent field properties and... more (see page 1871)
Handling Events (see page 1871)	Like most components, field components have events associated with them. Methods can be assigned as handlers for these events. By writing these handlers you can react to the occurrence of events that affect data entered in fields through data-aware controls and perform actions of your own design. The following table lists the events associated with field components: ***Field component events***
Accessing Field Values with the Default Dataset Property (see page 1872)	The most general method for accessing a field's value is to use Variants with the *FieldValues* property. For example, the following statement puts the value of an edit box into the *CustNo* field in the *Customers* table:
Defining a Calculated Field (see page 1872)	A calculated field displays values calculated at runtime by a dataset's OnCalcFields event handler. For example, you might create a string field that displays concatenated values from other fields.
Using Default Formatting for Numeric, Date, and Time Fields (see page 1873)	Delphi provides built-in display and edit format routines and intelligent default formatting for TFloatField, TCurrencyField, TBCDField, TFMTBCDField, TIntegerField, TSmallIntField, TWordField, TDateField, TDateTimeField, TTimeField, and TSQLTimeStampField components. To use these routines, you need do nothing. Default formatting is performed by the following routines: ***Field component formatting routines***
Setting Field Component Properties at Runtime (see page 1873)	You can use and manipulate the properties of field component at runtime. Access persistent field components by name, where the name can be obtained by concatenating the field name to the dataset name. For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *True*:

Creating Persistent Fields (see page 1874)	Persistent field components created with the Fields editor provide efficient, readable, and type-safe programmatic access to underlying data. Using persistent field components guarantees that each time your application runs, it always uses and displays the same columns, in the same order even if the physical structure of the underlying database has changed. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, Delphi generates an exception rather than running the application against a nonexistent column or mismatched data.
Converting Field Values (see page 1874)	Conversion properties attempt to convert one data type to another. For example, the *AsString* property converts numeric and Boolean values to string representations. The following table lists field component conversion properties, and which properties are recommended for field components by field-component class:
Dynamic Field Components (see page 1876)	Dynamically generated field components are the default. In fact, all field components for any dataset start out as dynamic fields the first time you place a dataset on a data module, specify how that dataset fetches its data, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the data represented by a dataset. Datasets generate one field component for each column in the underlying data. The exact TField descendant created for each column is determined by field type information received from the database or (for *TClientDataSet*) from a... more (see page 1876)
Working with Field Components: Overview (see page 1877)	Field components represent individual fields (columns) in datasets. You can use field components to control the display and editing of data in your applications. Field components are always associated with a dataset. You never use a TField object directly in your applications. Instead, each field component in your application is a *TField* descendant specific to the datatype of a column in a dataset. Field components provide data-aware controls such as TDBEdit and TDBGrid access to the data in a particular column of the associated dataset. Generally speaking, a single field component represents the characteristics of a single column, or field,... more (see page 1877)
Controlling and Masking User Input (see page 1878)	The *EditMask* property provides a way to control the type and range of values a user can enter into a data-aware component associated with TStringField, TDateField, TTimeField, and TDateTimeField, and TSQLTimeStampField components. You can use existing masks or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can, however, enter masks directly into the *EditMask* field in the **Object Inspector**. **Note:** For *TStringField* components, the *EditMask* property is also its display format.

3.2.3.14.1 Accessing Field Values with a Dataset's FieldByName Method

You can access the value of a field with a dataset's FieldByName method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion property, such as *AsString* or *AsInteger*. For example, the following statement assigns the value of the *CustNo* field in the *Customers* dataset to an edit control:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
Edit2->Text = Customers->FieldByName("CustNo")->AsString;
```

Conversely, you can assign a value to a field:

```
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
Customers->Edit();
Customers->FieldByName("CustNo")->AsString = Edit2->Text;
Customers->Post();
```

See Also

Displaying Field Component Values in Standard Controls (see page 1858)

Accessing Field Values with a Dataset's Fields Property (see page 1854)

3.2.3.14.2 Accessing Field Values with a Dataset's Fields Property

You can access the value of a field with the Fields property of the dataset component to which the field belongs. *Fields* maintains an indexed list of all the fields in the dataset. Accessing field values with the *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the *Fields* property you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using each field component's conversion properties (see page 1874).

For example, the following statement assigns the current value of the seventh column (Country) in the *Customers* table to an edit control:

```
Edit1.Text := CustTable.Fields[6].AsString;
Edit1->Text = CustTable->Fields->Fields[6]->AsString;
```

Conversely, you can assign a value to a field by setting the *Fields* property of the dataset to the desired field. For example:

```
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
Customers->Edit();
Customers->Insert();
Customers->Fields->Fields[6]->AsString = Edit1->Text;
Customers->Post();
```

See Also

Displaying Field Component Values in Standard Controls (see page 1858)

Accessing Field Values with a Dataset's FieldByName Method (see page 1853)

3.2.3.14.3 Arranging Persistent Fields

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of fields

1. Select the fields. You can select and order one or more fields at a time.
2. Drag the fields to a new location.

If you select a noncontiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block, the order of fields does not change.

Alternatively, you can select the field, and use `Ctrl+Up` and `Ctrl+Dn` to change an individual field's order in the list.

See Also

Defining New Persistent Fields (see page 1856)

Deleting Persistent Field Components (see page 1857)

3.2.3.14.4 Associating Attribute Sets with Field Components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*,

DisplayWidth, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), and you have saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to apply an attribute set.
3. Invoke the context menu and choose Associate Attributes.
4. Select or enter the attribute set to apply from the Associate Attributes dialog box. If there is an attribute set in the Data Dictionary that has the same name as the current field, that set name appears in the edit box.

   Warning: If the attribute set in the Data Dictionary is changed at a later date, you must reapply the attribute set to each field component that uses it. You can invoke the Fields editor and multi-select field components within a dataset when reapplying attributes.

See Also

Removing Attribute Associations (see page 1860)

The Data Dictionary (see page 1680)

3.2.3.14.5 Creating a Custom Constraint

A custom constraint is not imported from the server like other constraints. It is a constraint that you declare, implement, and enforce in your local application. As such, custom constraints can be useful for offering a prevalidation enforcement of data entry, but a custom constraint cannot be applied against data received from or sent to a server application.

To create a custom constraint, set the *CustomConstraint* property to specify a constraint condition, and set *ConstraintErrorMessage* to the message to display when a user violates the constraint at runtime.

CustomConstraint is an SQL string that specifies any application-specific constraints imposed on the field's value. Set *CustomConstraint* to limit the values that the user can enter into a field. *CustomConstraint* can be any valid SQL search expression such as

```
x > 0 and x < 100
```

The name used to refer to the value of the field can be any string that is not a reserved SQL keyword, as long as it is used consistently throughout the constraint expression.

Note: Custom constraints are only available in BDE-enabled and client datasets.

Custom constraints are imposed in addition to any constraints to the field's value that come from the server. To see the constraints imposed by the server, read the *ImportedConstraint* property.

See Also

Using Server Constraints (see page 1863)

3.2.3.14.6 Defining a Data Field

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a TSmallIntField with a TIntegerField. Because you cannot change a field's data type directly, you must define a new field to replace it.

Warning: Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a replacement data field for a field in a table underlying a dataset

1. Remove the field from the list of persistent fields assigned for the dataset, and then choose New Field from the context menu.
2. In the New Field dialog box, enter the name of an existing field in the database table in the Name edit box. Do not enter a new field name. You are actually specifying the name of the field from which your new field will derive its data.
3. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing. You cannot replace a string field of one size with a string field of another size. Note that while the data type should be different, it must be compatible with the actual data type of the field in the underlying table.
4. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type TStringField, TBytesField, and TVarBytesField.
5. Select Data in the Field type radio group if it is not already selected.
6. Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the **Object Inspector**. For more information about editing field component properties and events, see Setting persistent field properties and events (see page 1871).

See Also

Defining a Calculated Field (see page 1872)

Creating an Aggregate Field (see page 1856)

3.2.3.14.7 Defining an Aggregate Field

An aggregate field displays values from a maintained aggregate in a client dataset. An aggregate is a calculation that summarizes the data in a set of records. See Using maintained aggregates (see page 1735) for details about maintained aggregates.

To create an aggregate field in the New Field dialog box

1. Enter a name for the aggregate field in the Name edit box. Do not enter the name of an existing field.
2. Choose aggregate data type for the field from the Type combo box.
3. Select Aggregate in the Field type radio group.
4. Choose OK. The newly defined aggregate field is automatically added to the client dataset and its Aggregates property is automatically updated to include the appropriate aggregate specification.
5. Place the calculation for the aggregate in the *ExprText* property of the newly created aggregate field. For more information about defining an aggregate, see Specifying aggregates (see page 1730).

Once a persistent *TAggregateField* is created, a *TDBText* control can be bound to the aggregate field. The *TDBText* control will then display the value of the aggregate field that is relevant to the current record of the underlying client data set.

See Also

Defining a Data Field (see page 1855)

Defining a Calculated Field (see page 1872)

3.2.3.14.8 Defining New Persistent Fields

Besides making existing dataset fields into persistent fields, you can also create special persistent fields as additions to or replacements of the other persistent fields in a dataset.

New persistent fields that you create are only for display purposes. The data they contain at runtime are not retained either

because they already exist elsewhere in the database, or because they are temporary. The physical structure of the data underlying the dataset is not changed in any way.

To create a new persistent field component, invoke the context menu for the Fields editor by right clicking and choose New field. The New Field dialog box appears.

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

- The Field properties group box lets you enter general field component information. Enter the field name in the Name edit box. The name you enter here corresponds to the field component's *FieldName property. The New Field dialog uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's Name property and is only provided for informational purposes (Name is the identifier by which you refer to the field component in your source code). The dialog discards anything you enter directly in the Component edit box.*
- The Type combo box in the Field properties group lets you specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating-point currency values in a field, select *Currency* from the drop-down list. Use the Size edit box to specify the maximum number of characters that can be displayed or entered in a string-based field, or the size of *Bytes* and *VarBytes* fields. For all other data types, Size is meaningless.
- The Field type radio group lets you specify the type of new field component to create. The default type is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. You can also create Calculated fields, and if you are working with a client dataset, you can create InternalCalc fields or Aggregate fields. The following table describes these types of fields you can create:

Special persistent field kinds

Field Kind	Purpose
Data	Replaces an existing field (for example to change its data type)
Calculated	Displays values calculated at runtime by a dataset's *OnCalcFields* event handler.
Lookup	Retrieve values from a specified dataset at runtime based on search criteria you specify. (not supported by unidirectional datasets)
InternalCalc	Displays values calculated at runtime by a client dataset and stored with its data.
Aggregate	Displays a value summarizing the data in a set of records from a client dataset.

The Lookup definition group box is only used to create lookup fields. This is described more fully in Defining a lookup field (see page 1868).

The following topics describe how to create different field types:

- Defining a Data Field (see page 1855)
- Defining a Calculated Field (see page 1872)
- Defining a Lookup Field (see page 1868)
- Defining an Aggregate Field (see page 1856)

See Also

Arranging Persistent Fields (see page 1854)

Deleting Persistent Field Components (see page 1857)

3.2.3.14.9 Deleting Persistent Field Components

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table.

To remove one or more persistent field components for a dataset

1. Select the field(s) to remove in the Fields editor list box.

2. Press Del.

   Note: You can also delete selected fields by invoking the context menu and choosing Delete.

   Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always recreate a persistent field component that you delete by accident, but any changes previously made to its properties or events is lost. For more information, see Creating persistent fields (see page 1874)

   Note: If you remove all persistent field components for a dataset, the dataset reverts to using dynamic field components for every column in the underlying database table.

See Also

Arranging Persistent Fields (see page 1854)

Defining New Persistent Fields (see page 1856)

3.2.3.14.10 Displaying, Converting, and Accessing Field Values

Data-aware controls such as TDBEdit and TDBGrid automatically display the values associated with field components. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications. For more information about data-aware control, see Using data controls (see page 1778).

Standard controls can also display and edit database values associated with field components. Using standard controls, however, may require additional programming on your part. For example, when using standard controls, your application is responsible for tracking when to update controls because field values change. If the dataset has a datasource component, you can use its events to help you do this. In particular, the OnDataChange event lets you know when you may need to update a control's value and the OnStateChange event can help you determine when to enable or disable controls. For more information on these events, see Responding to changes mediated by the data source (see page 1776).

The following topics discuss how to work with field values so that you can display them in standard controls:

- Displaying Field Component Values in Standard Controls (see page 1858)
- Converting Field Values (see page 1874)
- Accessing Field Values with the Default Dataset Property (see page 1872)
- Accessing Field Values with a Dataset's Fields Property (see page 1854)
- Accessing Field Values with a Dataset's FieldByName Method (see page 1853)

See Also

Working with Field Component Methods at Runtime (see page 1864)

Persistent Field Components (see page 1859)

Setting a Default Value for a Field (see page 1860)

Working with Constraints (see page 1863)

Using Object Fields (see page 1862)

3.2.3.14.11 Displaying Field Component Values in Standard Controls

An application can access the value of a dataset column through the *Value* property of a field component. For example, the following *OnDataChange* event handler updates the text in a TEdit control because the value of the *CustomersCompany* field may have changed:

```
procedure TForm1.CustomersDataChange(Sender: TObject, Field: TField);
```

```
begin
  Edit3.Text := CustomersCompany.Value;
end;
void __fastcall TForm1::Table1DataChange(TObject *Sender, TField *Field)
{
  Edit3->Text = CustomersCompany->Value;
}
```

This method works well for string values, but may require additional programming to handle conversions for other data types. Fortunately, field components have built-in properties for handling conversions (see page 1874).

Note: You can also use Variants to access and set field values (see page 1872).

See Also

Accessing Field Values with a Dataset's Fields Property (see page 1854)

Accessing Field Values with a Dataset's FieldByName Method (see page 1853)

3.2.3.14.12 Persistent Field Components

By default, dataset fields are dynamic. Their properties and availability are automatically set and cannot be changed in any way. To gain control over a field's properties and events you must create persistent fields for the dataset. Persistent fields let you

- Set or change the field's display or edit characteristics at design time or runtime.
- Create new fields, such as lookup fields, calculated fields, and aggregated fields, that base their values on existing fields in a dataset.
- Validate data entry.
- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Define new fields to replace existing fields, based on columns in the table or query underlying a dataset.

At design time, you can—and should—use the Fields editor to create persistent lists of the field components used by the datasets in your application. Persistent field component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed. Once you create persistent fields with the Fields editor, you can also create event handlers for them that respond to changes in data values and that validate data entries.

Note: When you create persistent fields for a dataset, only those fields you select are available to your application at design time and runtime. At design time, you can always use the Fields editor to add or remove persistent fields for a dataset.

All fields used by a single dataset are either persistent or dynamic. You cannot mix field types in a single dataset. If you create persistent fields for a dataset, and then want to revert to dynamic fields, you must remove all persistent fields from the dataset. For more information about dynamic fields, see Dynamic field components (see page 1876).

Note: One of the primary uses of persistent fields is to gain control over the appearance and display of data. You can also control the appearance of columns in data-aware grids. To learn about controlling column appearance in grids, see Creating a customized grid (see page 1754).

The following topics describe how to use the Fields editor to create or modify the persistent fields in a dataset, and how to work with persistent fields:

- Creating Persistent Fields (see page 1874)
- Arranging Persistent Fields (see page 1854)
- Defining New Persistent Fields (see page 1856)
- Deleting Persistent Field Components (see page 1857)
- Setting Persistent Field Properties and Events (see page 1871)

See Also

Working with Field Component Methods at Runtime (see page 1864)

Displaying (see page 1858)

Setting a Default Value for a Field (see page 1860)

Working with Constraints (see page 1863)

Using Object Fields (see page 1862)

3.2.3.14.13 Removing Attribute Associations

If you change your mind about associating an attribute set with a field, you can remove the association.

To remove an attribute association

1. Invoke the Fields editor for the dataset containing the field.
2. Select the field or fields from which to remove the attribute association.
3. Invoke the context menu for the Fields editor and choose Unassociate Attributes.

   Warning: Unassociating an attribute set does not change any field properties. A field retains the settings it had when the attribute set was applied to it. To change these properties, select the field in the Fields editor and set its properties in the Object Inspector

   .

See Also

Associating Attribute Sets with Field Components (see page 1854)

The Data Dictionary (see page 1680)

3.2.3.14.14 Setting a Default Value for a Field

You can specify how a default value for a field in a client dataset or a BDE-enabled dataset should be calculated at runtime using the DefaultExpression property. *DefaultExpression* can be any valid SQL value expression that does not refer to field values. If the expression contains literals other than numeric values, they must appear in quotes. For example, a default value of noon for a time field would be

```
'12:00:00'
```

including the quotes around the literal value.

Note: If the underlying database table defines a default value for the field, the default you specify in *DefaultExpression* takes precedence. That is because *DefaultExpression* is applied when the dataset posts the record containing the field, before the edited record is applied to the database server.

See Also

Working with Field Component Methods at Runtime (see page 1864)

Displaying (see page 1858)

Persistent Field Components (see page 1859)

Working with Constraints (see page 1863)

Using Object Fields (see page 1862)

3.2.3.14.15 Setting Display and Edit Properties at Design Time

To edit the display properties of a selected field component, switch to the Properties page on the **Object Inspector** window. The

following table summarizes display properties that can be edited.

Field component properties

Property	Purpose
Alignment	Left justifies, right justifies, or centers a field contents within a data-aware component.
ConstraintErrorMessage	Specifies the text to display when edits clash with a constraint condition.
CustomConstraint	Specifies a local constraint to apply to data during editing.
Currency	Numeric fields only. *True*: displays monetary values.False (default): does not display monetary values.
DisplayFormat	Specifies the format of data displayed in a data-aware component.
DisplayLabel	Specifies the column name for a field in a data-aware grid component.
DisplayWidth	Specifies the width, in characters, of a grid column that display this field.
EditFormat	Specifies the edit format of data in a data-aware component.
EditMask	Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear within the field (hyphens, parentheses, and so on).
FieldKind	Specifies the type of field to create.
FieldName	Specifies the actual name of a column in the table from which the field derives its value and data type.
HasConstraints	Indicates whether there are constraint conditions imposed on a field.
ImportedConstraint	Specifies an SQL constraint imported from the Data Dictionary or an SQL server.
Index	Specifies the order of the field in a dataset.
LookupDataSet	Specifies the table used to look up field values when *Lookup* is *True*.
LookupKeyFields	Specifies the field(s) in the lookup dataset to match when doing a lookup.
LookupResultField	Specifies the field in the lookup dataset from which to copy values into this field.
MaxValue	Numeric fields only. Specifies the maximum value a user can enter for the field.
MinValue	Numeric fields only. Specifies the minimum value a user can enter for the field.
Name	Specifies the component name of the field component within Delphi.
Origin	Specifies the name of the field as it appears in the underlying database.
Precision	Numeric fields only. Specifies the number of significant digits.
ReadOnly	*True*: Displays field values in data-aware controls, but prevents editing. *False* (the default): Permits display and editing of field values.
Size	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of *TBytesField* and *TVarBytesField* fields.
Tag	Long integer bucket available for programmer use in every component as needed.
Transliterate	*True* (default): specifies that translation to and from the respective locales will occur as data is transferred between a dataset and a database. *False*: Locale translation does not occur.
Visible	*True* (the default): Permits display of field in a data-aware grid.False: Prevents display of field in a data-aware grid component.User-defined components can make display decisions based on this property.

Not all properties are available for all field components. For example, a field component of type TStringField does not have *Currency*, *MaxValue*, or *DisplayFormat* properties, and a component of type TFloatField does not have a *Size* property.

While the purpose of most properties is straightforward, some properties, such as Calculated, require additional programming

steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see Controlling and masking user input (see page 1878).

3.2.3.14.16 Using Object Fields

Object fields are fields that represent a composite of other, simpler datatypes. These include ADT (Abstract Data Type) fields, Array fields, DataSet fields, and Reference fields. All of these field types either contain or reference child fields or other data sets.

ADT fields and array fields are fields that contain child fields. The child fields of an ADT field can be any scalar or object type (that is, any other field type). These child fields may differ in type from each other. An array field contains an array of child fields, all of the same type.

Dataset and reference fields are fields that access other data sets. A dataset field provides access to a nested (detail) dataset and a reference field stores a pointer (reference) to another persistent object (ADT).

Types of object field components

Component Name	Purpose
TADTField	Represents an ADT (Abstract Data Type) field.
TArrayField	Represents an array field.
TDataSetField	Represents a field that contains a nested data set reference.
TReferenceField	Represents a REF field, a pointer to an ADT.

When you add fields with the Fields editor to a dataset that contains object fields, persistent object fields of the correct type are automatically created for you. Adding persistent object fields to a dataset automatically sets the dataset's ObjectView property to *True*, which instructs the dataset to store these fields hierarchically, rather than flattening them out as if the constituent child fields were separate, independent fields.

The following properties are common to all object fields and provide the functionality to handle child fields and datasets.

Common object field descendant properties

Property	Purpose
Fields	Contains the child fields belonging to the object field.
ObjectType	Classifies the object field.
FieldCount	Number of child fields belonging to the object field.
FieldValues	Provides access to the values of the child fields.

Displaying ADT and array fields

Both ADT and array fields contain child fields that can be displayed through data-aware controls.

Data-aware controls such as TDBEdit that represent a single field value display child field values in an uneditable comma delimited string. In addition, if you set the control's *DataField* property to the child field instead of the object field itself, the child field can be viewed an edited just like any other normal data field.

A TDBGrid control displays ADT and array field data differently, depending on the value of the dataset's ObjectView property. When *ObjectView* is *False*, each child field appears in a single column. When *ObjectView* is *True*, an ADT or array field can be expanded and collapsed by clicking on the arrow in the title bar of the column. When the field is expanded, each child field appears in its own column and title bar, all below the title bar of the ADT or array itself. When the ADT or array is collapsed, only one column appears with an uneditable comma-delimited string containing the child fields.

The following topics discuss each type of object field in more detail:

- Working with ADT Fields (see page 1864)
- Working with Array Fields (see page 1866)
- Working with Dataset Fields (see page 1867)
- Working with Reference Fields (see page 1867)

See Also

Working with Field Component Methods at Runtime (see page 1864)

Displaying (see page 1858)

Setting a Default Value for a Field (see page 1860)

Working with Constraints (see page 1863)

Persistent Field Components (see page 1859)

3.2.3.14.17 Using Server Constraints

Most production SQL databases use constraints to impose conditions on the possible values for a field. For example, a field may not permit NULL values, may require that its value be unique for that column, or that its values be greater than *0* and less than *150*. While you could replicate such conditions in your client applications, client datasets and BDE-enabled datasets offer the *ImportedConstraint* property to propagate a server's constraints locally.

ImportedConstraint is a read-only property that specifies an SQL clause that limits field values in some manner. For example:

```
Value > 0 and Value < 100
```

Do not change the value of *ImportedConstraint*, except to edit nonstandard or server-specific SQL that has been imported as a comment because it cannot be interpreted by the database engine.

To add additional constraints on the field value, use the *CustomConstraint* property. Custom constraints are imposed in addition to the imported constraints. If the server constraints change, the value of *ImportedConstraint* also changed but constraints introduced in the *CustomConstraint* property persist.

Removing constraints from the *ImportedConstraint* property will not change the validity of field values that violate those constraints. Removing constraints results in the constraints being checked by the server instead of locally. When constraints are checked locally, the error message supplied as the *ConstraintErrorMessage* property is displayed when violations are found, instead of displaying an error message from the server.

See Also

Creating a Custom Constraint (see page 1855)

3.2.3.14.18 Working with Constraints

Field components in client datasets or BDE-enabled datasets can use SQL server constraints (see page 1863). In addition, your applications can create and use custom constraints (see page 1855) for these datasets that are local to your application. All constraints are rules or conditions that impose a limit on the scope or range of values that a field can store.

See Also

Working with Field Component Methods at Runtime (see page 1864)

Displaying (see page 1858)

Setting a Default Value for a Field (see page 1860)

Persistent Field Components (see page 1859)

Using Object Fields (see page 1862)

3.2.3.14.19 Working with Field Component Methods at Runtime

Field components methods available at runtime enable you to convert field values from one data type to another, and enable you to set focus to the first data-aware control in a form that is associated with a field component.

Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as BeforePost). Validation may be performed on the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control containing the faulty data to have focus so that the user can enter corrections.

You control focus for a field's data-aware components with a field's FocusControl method. *FocusControl* sets focus to the first data-aware control in a form that is associated with a field. An event handler should call a field's *FocusControl* method before validating the field. The following code illustrates how to call the *FocusControl* method for the *Company* field in the *Customers* table:

```
CustomersCompany.FocusControl;
CustomersCompany->FocusControl();
```

The following table lists some other field component methods and their uses. For a complete list and detailed information about using each method, see TField.

Selected field component methods

Method	Purpose
AssignValue	Sets a field value to a specified value using an automatic conversion function based on the field's type.
Clear	Clears the field and sets its value to NULL.
GetData	Retrieves unformatted data from the field.
IsValidChar	Determines if a character entered by a user in a data-aware control to set a value is allowed for this field.
SetData	Assigns unformatted data to this field.

See Also

Persistent Field Components (see page 1859)

Displaying (see page 1858)

Setting a Default Value for a Field (see page 1860)

Working with Constraints (see page 1863)

Using Object Fields (see page 1862)

3.2.3.14.20 Working with ADT Fields

ADTs are user-defined types created on the server, and are similar to the record type. An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs.

There are a variety of ways to access the data in ADT field types. These are illustrated in the following examples, which assign a child field value to an edit box called CityEdit, and use the following ADT structure,

```
Address
  Street
  City
```

```
  State
  Zip
```

Using persistent field components

The easiest way to access ADT field values is to use persistent field components. For the ADT structure above, the following persistent fields can be added to the Customer table using the Fields editor:

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

Given these persistent fields, you can simply access the child fields of an ADT field by name:

```
CityEdit.Text := CustomerAddrCity.AsString;
CityEdit->Text = CustomerAddrCity->AsString;
```

Although persistent fields are the easiest way to access ADT child fields, it is not possible to use them if the structure of the dataset is not known at design time. When accessing ADT child fields without using persistent fields, you must set the dataset's ObjectView property to True.

Using the dataset's FieldByName method

You can access the children of an ADT field using the dataset's FieldByName method by qualifying the name of the child field with the ADT field's name:

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

Using the dateset's FieldValues property

You can also use qualified field names with a dataset's *FieldValues* property:

```
CityEdit.Text := Customer['Address.City'];
CityEdit->Text = Customer->FieldValues["Address.City"];
```

Note that you can omit the property name (*FieldValues*) because *FieldValues* is the dataset's default property.

Note: Unlike other runtime methods for accessing ADT child field values, the *FieldValues* property works even if the dataset's *ObjectView* property is False.

Using the ADT field's FieldValues property

You can access the value of a child field with the TADTField's FieldValues property. FieldValues accepts and returns a Variant, so it can handle and convert fields of any type. The index parameter is an integer value that specifies the offset of the field.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->FieldValues[1];
```

Because FieldValues is the default property of TADTField, the property name (FieldValues) can be omitted. Thus, the following statement is equivalent to the one above:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

Using the ADT field's Fields property

Each ADT field has a Fields property that is analogous to the *Fields* property of a dataset. Like the *Fields* property of a dataset, you can use it to access child fields by position:

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->Fields->
```

or by name:

```
CityEdit.Text :=
TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
CityEdit->Text = ((TADTField*)Customer->FieldByName("Address"))->
CityEdit->Text = Customer->FieldByName("Address.City")->AsString;
```

See Also

Working with Array Fields (see page 1866)

Working with DataSet Fields (see page 1867)

Working with Reference Fields (see page 1867)

3.2.3.14.21 Working with Array Fields

Array fields consist of a set of fields of the same type. The field types can be scalar (for example, float, string), or non-scalar (an ADT), but an array field of arrays is not permitted. The SparseArrays property of *TDataSet* determines whether a unique TField object is created for each element of the array field.

There are a variety of ways to access the data in array field types. If you are not using persistent fields, the dataset's ObjectView property must be set to True before you can access the elements of an array field.

Using persistent fields

You can map persistent fields to the individual array elements in an array field. For example, consider an array field *TelNos_Array*, which is a six element array of strings. The following persistent fields created for the Customer table component represent the *TelNos_Array* field and its six elements:

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
CustomerTELNOS_ARRAY: TArrayField;
CustomerTELNOS_ARRAY0: TStringField;
CustomerTELNOS_ARRAY1: TStringField;
CustomerTELNOS_ARRAY2: TStringField;
CustomerTELNOS_ARRAY3: TStringField;
CustomerTELNOS_ARRAY4: TStringField;
CustomerTELNOS_ARRAY5: TStringField;
```

Given these persistent fields, the following code uses a persistent field to assign an array element value to an edit box named *TelEdit*.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
TelEdit->Text = CustomerTELNOS_ARRAY0->AsString;
```

Using the array field's FieldValues property

You can access the value of a child field with the array field's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert child fields of any type. For example,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
TelEdit->Text = ((TArrayField*)Customer->FieldByName("TelNos_Array"))->FieldValues[1];
```

Because *FieldValues* is the default property of *TArrayField*, this can also be written

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

Using the array field's Fields property

TArrayField has a Fields property that you can use to access individual sub-fields. This is illustrated below, where an array field (*OrderDates*) is used to populate a list box with all non-null array elements:

```
for I := 0 to OrderDates.Size - 1 do
begin
  if not OrderDates.Fields[I].IsNull then
    OrderDateListBox.Items.Add(OrderDates[I]);
```

```
end;
for (int i = 0; i < OrderDates->Size; ++i)
  if (!OrderDates->Fields->Fields[i]->IsNull)
    OrderDateListBox->Items->Add(OrderDates->Fields->Fields[i]->AsString);
```

See Also

Working with ADT Fields (see page 1864)

Working with DataSet Fields (see page 1867)

Working with Reference Fields (see page 1867)

3.2.3.14.22 Working with DataSet Fields

Dataset fields provide access to data stored in a nested dataset. The NestedDataSet property references the nested dataset. The data in the nested dataset is then accessed through the field objects of the nested dataset.

Displaying dataset fields

TDBGrid controls enable the display of data stored in data set fields. In a TDBGrid control, a dataset field is indicated in each cell of a dataset column with the string "(DataSet)", and at runtime an ellipsis button also exists to the right. Clicking on the ellipsis brings up a new form with a grid displaying the dataset associated with the current record's dataset field. This form can also be brought up programmatically with the DB grid's ShowPopupEditor method. For example, if the seventh column in the grid represents a dataset field, the following code will display the dataset associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

Accessing data in a nested dataset

A dataset field is not normally bound directly to a data aware control. Rather, since a nested data set is just that, a data set, the means to get at its data is via a *TDataSet* descendant. The type of dataset you use is determined by the parent dataset (the one with the dataset field.) For example, a BDE-enabled dataset uses *TNestedTable* to represent the data in its dataset fields, while client datasets use other client datasets.

To access the data in a dataset field

1. Create a persistent TDataSetField object by invoking the Fields editor for the parent dataset.
2. Create a dataset to represent the values in that dataset field. It must be of a type compatible with the parent dataset.
3. Set that DataSetField property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the nested dataset field for the current record has a value, the detail dataset component will contain records with the nested data; otherwise, the detail dataset will be empty.

Before inserting records into a nested dataset, you should be sure to post the corresponding record in the master table, if it has just been inserted. If the inserted record is not posted, it will be automatically posted before the nested dataset posts.

See Also

Working with ADT Fields (see page 1864)

Working with Array Fields (see page 1866)

Working with Reference Fields (see page 1867)

3.2.3.14.23 Working with Reference Fields

Reference fields store a pointer or reference to another ADT object. This ADT object is a single record of another object table. Reference fields always refer to a single record in a dataset (object table). The data in the referenced object is actually returned in a nested dataset, but can also be accessed via the Fields property on the *TReferenceField*.

Displaying reference fields

In a TDBGrid control a reference field is designated in each cell of the dataset column, with (Reference) and, at runtime, an ellipsis button to the right. At runtime, clicking on the ellipsis brings up a new form with a grid displaying the object associated with the current record's reference field.

This form can also be brought up programmatically with the DB grid's ShowPopupEditor method. For example, if the seventh column in the grid represents a reference field, the following code will display the object associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
DBGrid1->ShowPopupEditor(DBGrid1->Columns->Items[7], -1, -1);
```

Accessing data in a reference field

You can access the data in a reference field in the same way you access a nested dataset.

To access data in a reference field

1. Create a persistent TDataSetField object by invoking the Fields editor for the parent dataset.
2. Create a dataset to represent the value of that dataset field.
3. Set that DataSetField property of the dataset created in step 2 to the persistent dataset field you created in step 1.

If the reference is assigned, the reference dataset will contain a single record with the referenced data. If the reference is null, the reference dataset will be empty.

You can also use the reference field's Fields property to access the data in a reference field. For example, the following lines are equivalent and assign data from the reference field *CustomerRefCity* to an edit box called *CityEdit*:

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
CityEdit->Text = CustomerADDRESS_REF->NestedDataSet->Fields->Fields[1]->AsString;
```

When data in a reference field is edited, it is actually the referenced data that is modified.

To assign a reference field, you need to first use a SELECT statement to select the reference from the table, and then assign. For example:

```
var
  AddressQuery: TQuery;
  CustomerAddressRef: TReferenceField;
begin
  AddressQuery.SQL.Text := 'SELECT REF(A) FROM AddressTable A WHERE A.City = ''San
Francisco''';
  AddressQuery.Open;
  CustomerAddressRef.Assign(AddressQuery.Fields[0]);
end;
AddressQuery->SQL->Text = "SELECT REF(A) FROM AddressTable A WHERE A.City = "San Francisco"";
AddressQuery->Open();
CustomerAddressRef->Assign(AddressQuery->Fields->Fields[0]);
```

See Also

Working with ADT Fields (see page 1864)

Working with Array Fields (see page 1866)

Working with DataSet Fields (see page 1867)

3.2.3.14.24 Defining a Lookup Field

A lookup field is a read-only field that displays values at runtime based on search criteria you specify. In its simplest form, a lookup field is passed the name of an existing field to search on, a field value to search for, and a different field in a lookup

dataset whose value it should display.

For example, consider a mail-order application that enables an operator to use a lookup field to determine automatically the city and state that correspond to the zip code a customer provides. The column to search on might be called *ZipTable.Zip*, the value to search for is the customer's zip code as entered in *Order.CustZip*, and the values to return would be those for the *ZipTable.City* and *ZipTable.State* columns of the record where the value of *ZipTable.Zip* matches the current value in the *Order.CustZip* field.

Note: Unidirectional datasets do not support lookup fields.

To create a lookup field in the New Field dialog box

1. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type TStringField, TBytesField, and TVarBytesField.
4. Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.
5. Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.
6. Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons. If you are using more than one field, you must use persistent field components.
7. Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. If you specified more than one key field, you must specify the same number of lookup keys. To specify more than one field, enter field names directly, separating multiple field names with semicolons.
8. Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating.

When you design and run your application, lookup field values are determined before calculated field values are calculated. You can base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields.

You can use the LookupCache property to hone the way lookup fields are determined. *LookupCache* determines whether the values of a lookup field are cached in memory when a dataset is first opened, or looked up dynamically every time the current record in the dataset changes. Set *LookupCache* to *True* to cache the values of a lookup field when the LookupDataSet is unlikely to change and the number of distinct lookup values is small. Caching lookup values can speed performance, because the lookup values for every set of *LookupKeyFields values are preloaded when the DataSet is opened. When the current record in the DataSet changes, the field object can locate its Value in the cache, rather than accessing the LookupDataSet. This performance improvement is especially dramatic if the LookupDataSet is on a network where access is slow.*

If every record of DataSet has different values for KeyFields, the overhead of locating values in the cache can be greater than any performance benefit provided by the cache. The overhead of locating values in the cache increases with the number of distinct values that can be taken by KeyFields.

If LookupDataSet is volatile, caching lookup values can lead to inaccurate results. Call RefreshLookupList to update the values in the lookup cache. RefreshLookupList regenerates the LookupList property, which contains the value of the LookupResultField for every set of LookupKeyFields values.

When setting LookupCache at runtime, call RefreshLookupList to initialize the cache.

See Also

Defining a Data Field (see page 1855)

Defining a Calculated Field (see page 1872)

Creating an Aggregate Field (see page 1856)

3.2.3.14.25 Creating Attribute Sets for Field Components

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields.

Note: Attribute sets and the Data Dictionary are only available for BDE-enabled datasets.

To create an attribute set based on a field component in a dataset

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to set properties.
3. Set the desired properties for the field in the **Object Inspector**.
4. Right-click the Fields editor list box to invoke the context menu.
5. Choose Save Attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing Save Attributes As instead of Save Attributes from the context menu.

Once you have created a new attribute set and added it to the Data Dictionary, you can then associate it with other persistent field components (see page 1854). Even if you later remove the association (see page 1860), the attribute set remains defined in the Data Dictionary.

See Also

Setting Display and Edit Properties at Design Time (see page 1860)

Associating Attribute Sets with Field Components (see page 1854)

Removing Attribute Associations (see page 1860)

The Data Dictionary (see page 1680)

3.2.3.14.26 Programming a Calculated Field

After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.

To program a value for a calculated field

1. Select the dataset component from the **Object Inspector** drop-down list.
2. Choose the **Object Inspector** Events page.
3. Double-click the OnCalcFields property to bring up or create a CalcFields procedure for the dataset component.
4. Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a *CityStateZip* calculated field for the *Customers* table on the *CustomerData* data module. *CityStateZip* should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* procedure for the *Customers* table, select the *Customers* table from the **Object Inspector** drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The *TCustomerData.CustomersCalcFields* procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value  + ' ' +
CustomersZip.Value;
CustomersCityStateZip->Value = CustomersCity->Value + AnsiString(", ") +
```

Note: When writing the *OnCalcFields* event handler for an internally calculated field, you can improve performance by checking the client dataset's State property and only recomputing the value when *State* is *dsInternalCalc*. See Using internally calculated fields in client datasets (see page 1735) for details.

See Also

Defining a Calculated Field (see page 1872)

3.2.3.14.27 Setting Persistent Field Properties and Events

You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a TDBGrid, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the **Object Inspector**.

The following topics discuss using persistent field properties and events:

- Setting Display and Edit Properties at Design Time (see page 1860)
- Setting Field Component Properties at Runtime (see page 1873)
- Creating Attribute Sets for Field Components (see page 1870)
- Controlling and Masking User Input (see page 1878)
- Using Default Formatting for Numeric (see page 1873)
- Handling Events (see page 1871)

See Also

Arranging Persistent Fields (see page 1854)

Defining New Persistent Fields (see page 1856)

Deleting Persistent Field Components (see page 1857)

3.2.3.14.28 Handling Events

Like most components, field components have events associated with them. Methods can be assigned as handlers for these events. By writing these handlers you can react to the occurrence of events that affect data entered in fields through data-aware controls and perform actions of your own design. The following table lists the events associated with field components:

Field component events

Event	Purpose
OnChange	Called when the value for a field changes.
OnGetText	Called when the value for a field component is retrieved for display or editing.
OnSetText	Called when the value for a field component is set.
OnValidate	Called to validate the value for a field component whenever the value is changed because of an edit or insert operation.

OnGetText and OnSetText events are primarily useful to programmers who want to do custom formatting that goes beyond the built-in formatting functions. *OnChange* is useful for performing application-specific tasks associated with data change, such as enabling or disabling menus or visual controls. OnValidate is useful when you want to control data-entry validation in your application before returning values to a database server.

To write an event handler for a field component

1. Select the component.
2. Select the Events page in the **Object Inspector**.
3. Double-click the Value field for the event handler to display its source code window.
4. Create or edit the handler code.

See Also

Setting Display and Edit Properties at Design Time (see page 1860)

3.2.3.14.29 Accessing Field Values with the Default Dataset Property

The most general method for accessing a field's value is to use Variants with the *FieldValues* property. For example, the following statement puts the value of an edit box into the *CustNo* field in the *Customers* table:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
Customers->FieldValues["CustNo"] = Edit2->Text;
```

Because the *FieldValues* property is of type *Variant*, it automatically converts other datatypes into a *Variant* value.

See Also

Displaying Field Component Values in Standard Controls (see page 1858)

Accessing Field Values with a Dataset's Fields Property (see page 1854)

Accessing Field Values with a Dataset's FieldByName Method (see page 1853)

3.2.3.14.30 Defining a Calculated Field

A calculated field displays values calculated at runtime by a dataset's OnCalcFields event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box

1. Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type TStringField, TBytesField, and TVarBytesField.
4. Select Calculated or InternalCalc in the Field type radio group. InternalCalc is only available if you are working with a client dataset. The significant difference between these types of calculated fields is that the values calculated for an InternalCalc field are stored and retrieved as part of the client dataset's data.
5. Choose OK. The newly defined calculated field is automatically added to the end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's or data module's **type** declaration.
6. Place code that calculates values for the field in the OnCalcFields event handler for the dataset. For more information about writing code to calculate field values, see Programming a calculated field (see page 1870).

   Note: To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector

   . For more information about editing field component properties and events, see Setting persistent field properties and events (see page 1871).

See Also

Defining a Data Field (see page 1855)

Creating an Aggregate Field (see page 1856)

Programming a Calculated Field (see page 1870)

3.2.3.14.31 Using Default Formatting for Numeric, Date, and Time Fields

Delphi provides built-in display and edit format routines and intelligent default formatting for TFloatField, TCurrencyField, TBCDField, TFMTBCDField, TIntegerField, TSmallIntField, TWordField, TDateField, TDateTimeField, TTimeField, and TSQLTimeStampField components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

Field component formatting routines

Routine	Used by . . .
FormatFloat	*TFloatField*, *TCurrencyField*
FormatDateTime	*TDateField*, *TTimeField*, *TDateTimeField*,
SQLTimeStampToString	*TSQLTimeStampField*
FormatCurr	*TCurrencyField, TBCDField*
BcdToStrF	TFMTBCDField

Only format properties appropriate to the data type of a field component are available for a given component.

Default formatting conventions for date, time, currency, and numeric values are based on the Regional Settings properties in the Control Panel. For example, using the default settings for the United States, a *TFloatField* column with the *Currency* property set to *True* sets the *DisplayFormat* property for the value 1234.56 to $1234.56, while the EditFormat is 1234.56.

At design time or runtime, you can edit the *DisplayFormat* and EditFormat properties of a field component to override the default display settings for that field. You can also write *OnGetText and OnSetText event handlers to do custom formatting for field components at runtime.*

See Also

Setting Display and Edit Properties at Design Time (see page 1860)

3.2.3.14.32 Setting Field Component Properties at Runtime

You can use and manipulate the properties of field component at runtime. Access persistent field components by name, where the name can be obtained by concatenating the field name to the dataset name.

For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *True*:

```
CustomersCityStateZip.ReadOnly := True;
CustomersCityStateZip->ReadOnly = true;
```

And this statement changes field ordering by setting the Index property of the *CityStateZip* field in the *Customers* table to 3:

```
CustomersCityStateZip.Index := 3;
CustomersCityStateZip->Index = 3;
```

See Also

Setting Display and Edit Properties at Design Time (see page 1860)

3.2.3.14.33 Creating Persistent Fields

Persistent field components created with the Fields editor provide efficient, readable, and type-safe programmatic access to underlying data. Using persistent field components guarantees that each time your application runs, it always uses and displays the same columns, in the same order even if the physical structure of the underlying database has changed. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, Delphi generates an exception rather than running the application against a nonexistent column or mismatched data.

To create persistent fields for a dataset

1. Place a dataset in a data module.
2. Bind the dataset to its underlying data. This typically involves associating the dataset with a connection component or provider and specifying any properties to describe the data. For example, If you are using *TADODataSet*, you can set the *Connection* property to a properly configured *TADOConnection* component and set the *CommandText* property to a valid query.
3. Double-click the dataset component in the data module to invoke the Fields editor. The Fields editor contains a title bar, navigator buttons, and a list box. The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. For example, if you open the *Customers* dataset in the *CustomerData* data module, the title bar displays 'CustomerData.Customers,' or as much of the name as fits. Below the title bar is a set of navigation buttons that let you scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty. If the dataset is unidirectional, the buttons for moving to the last record and the previous record are always dimmed. The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.
4. Right click in the Fields editor and choose Add Fields.
5. Select the fields to make persistent in the Add Fields dialog box. By default, all fields are selected when the dialog box opens. Any fields you select become persistent fields.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and (if the dataset is not unidirectional) Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, it no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, it verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, the dataset raises an exception warning you that the field is not valid, and does not open the dataset.

See Also

Arranging Persistent Fields (see page 1854)

Defining New Persistent Fields (see page 1856)

Deleting Persistent Field Components (see page 1857)

3.2.3.14.34 Converting Field Values

Conversion properties attempt to convert one data type to another. For example, the *AsString* property converts numeric and Boolean values to string representations. The following table lists field component conversion properties, and which properties are recommended for field components by field-component class:

Field Components	AsVariant	AsString	AsInteger	AsFloat, AsCurrency, AsBCD	AsDateTime, AsSQLTimeStamp	AsBoolean
TStringField	yes	NA	yes	yes	yes	yes
TWideStringField	yes	yes	yes	yes	yes	yes
TIntegerField	yes	yes	NA	yes		
TSmallIntField	yes	yes	yes	yes		
TWordField	yes	yes	yes	yes		
TLargeintField	yes	yes	yes	yes		
TFloatField	yes	yes	yes	yes		
TCurrencyField	yes	yes	yes	yes		
TBCDField	yes	yes	yes	yes		
TFMTBCDField	yes	yes	yes	yes		
TDateTimeField	yes	yes		yes	yes	
TDateField	yes	yes		yes	yes	
TTimeField	yes	yes		yes	yes	
TSQLTimeStampField	yes	yes		yes	yes	
TBooleanField	yes	yes				
TBytesField	yes	yes				
TVarBytesField	yes	yes				
TBlobField	yes	yes				
TMemoField	yes	yes				
TGraphicField	yes	yes				
TVariantField	NA	yes	yes	yes	yes	yes
TAggregateField	yes	yes				

Note that some columns in the table refer to more than one conversion property (such as *AsFloat*, *AsCurrency*, and *AsBCD*). This is because all field data types that support one of those properties always support the others as well.

Note also that the *AsVariant* property can translate among all data types. For any datatypes not listed above, *AsVariant* is also available (and is, in fact, the only option). When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a TDateTimeField value into a float format? *AsFloat* converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. The following table lists permissible conversions that produce special results:

Special conversion results

Conversion	Result
String to Boolean	Converts "True," "False," "Yes," and "No" to Boolean. Other values raise exceptions.
Float to Integer	Rounds float value to nearest integer value.

DateTime or SQLTimeStamp to Float	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
Boolean to String	Converts any Boolean value to "True" or "False."

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

Conversion always occurs before an assignment is made. For example, the following statement converts the value of *CustomersCustNo* to a string and assigns the string to the text of an edit control:

```
Edit1.Text := CustomersCustNo.AsString;
Edit1->Text = CustomersCustNo->AsString;
```

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
MyTableMyField->AsInteger = StrToInt(Edit1->Text);
```

See Also

Displaying Field Component Values in Standard Controls (see page 1858)

Accessing Field Values with a Dataset's Fields Property (see page 1854)

Accessing Field Values with a Dataset's FieldByName Method (see page 1853)

3.2.3.14.35 Dynamic Field Components

Dynamically generated field components are the default. In fact, all field components for any dataset start out as dynamic fields the first time you place a dataset on a data module, specify how that dataset fetches its data, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the data represented by a dataset. Datasets generate one field component for each column in the underlying data. The exact TField descendant created for each column is determined by field type information received from the database or (for *TClientDataSet*) from a provider component.

Dynamic fields are temporary. They exist only as long as a dataset is open. Each time you reopen a dataset that uses dynamic fields, it rebuilds a completely new set of dynamic field components based on the current structure of the data underlying the dataset. If the columns in the underlying data change, then the next time you open a dataset that uses dynamic field components, the automatically generated field components are also changed to match.

Use dynamic fields in applications that must be flexible about data display and editing. For example, to create a database browsing tool you must use dynamic fields because every database table has different numbers and types of columns. You might also want to use dynamic fields in applications where user interaction with data mostly takes place inside grid components and you know that the datasets used by the application change frequently.

To use dynamic fields in an application

1. Place datasets and data sources in a data module.
2. Associate the datasets with data. This involves using a connection component or provider to connect to the source of the data and setting any properties that specify what data the dataset represents.
3. Associate the data sources with the datasets.
4. Place data-aware controls in the application's forms, add the data module to each uses clause for each form's unit, and associate each data-aware control with a data source in the module. In addition, associate a field with each data-aware control that requires one. Note that because you are using dynamic field components, there is no guarantee that any field name you specify will exist when the dataset is opened.
5. Open the datasets.

Aside from ease of use, dynamic fields can be limiting. Without writing code, you cannot change the display and editing defaults for dynamic fields, you cannot safely change the order in which dynamic fields are displayed, and you cannot prevent access

to any fields in the dataset. You cannot create additional fields for the dataset, such as calculated fields or lookup fields, and you cannot override a dynamic field's default data type. To gain control and flexibility over fields in your database applications, you need to invoke the Fields editor to create persistent field components for your datasets.

See Also

Persistent Field Components (see page 1859)

Working with Field Component Methods at Runtime (see page 1864)

Displaying (see page 1858)

Setting a Default Value for a Field (see page 1860)

Working with Constraints (see page 1863)

Using Object Fields (see page 1862)

3.2.3.14.36 Working with Field Components: Overview

Field components represent individual fields (columns) in datasets. You can use field components to control the display and editing of data in your applications.

Field components are always associated with a dataset. You never use a TField object directly in your applications. Instead, each field component in your application is a *TField* descendant specific to the datatype of a column in a dataset. Field components provide data-aware controls such as TDBEdit and TDBGrid access to the data in a particular column of the associated dataset.

Generally speaking, a single field component represents the characteristics of a single column, or field, in a dataset, such as its data type and size. It also represents the field's display characteristics, such as alignment, display format, and edit format. For example, a TFloatField component has four properties that directly affect the appearance of its data:

TFloatField properties that affect data display

Property	Purpose
Alignment	Specifies whether data is displayed left-aligned, centered, or right-aligned.
DisplayWidth	Specifies the number of digits to display in a control at one time.
DisplayFormat	Specifies data formatting for display (such as how many decimal places to show).
EditFormat	Specifies how to display a value during editing.

As you scroll from record to record in a dataset, a field component lets you view and change the value for that field in the current record.

Field components have many properties in common with one another (such as DisplayWidth and *Alignment*), and they have properties specific to their data types (such as Precision for *TFloatField*). Each of these properties affect how data appears to an application's users on a form. Some properties, such as *Precision*, can also affect what data values the user can enter in a control when modifying or entering data.

All field components for a dataset are either *dynamic* (automatically generated for you based on the underlying structure of database tables), or *persistent* (generated based on specific field names and properties you set in the Fields editor). Dynamic and persistent fields have different strengths and are appropriate for different types of applications.

The following topics discuss field components in greater detail:

- Dynamic Field Components (see page 1876)
- Persistent Field Components (see page 1859)
- Working with Field Component Methods at Runtime (see page 1864)

- Displaying (see page 1858)
- Setting a Default Value for a Field (see page 1860)
- Working with Constraints (see page 1863)
- Using Object Fields (see page 1862)

See Also

Understanding Datasets (see page 1632)

Designing Database Applications (see page 1566)

Using Data Controls (see page 1778)

Creating and Using a Client Dataset (see page 1740)

Connecting to Databases (see page 1506)

Working With ADO Components (see page 1494)

Using Unidirectional Datasets (see page 1823)

Using the Borland Database Engine (see page 1643)

Using Provider Components (see page 1819)

Creating Multi-tiered Applications: Overview (see page 1518)

Using Decision Support Components (see page 1800)

Using XML in Database Applications (see page 1847)

3.2.3.14.37 Controlling and Masking User Input

The *EditMask* property provides a way to control the type and range of values a user can enter into a data-aware component associated with TStringField, TDateField, TTimeField, and TDateTimeField, and TSQLTimeStampField components. You can use existing masks or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can, however, enter masks directly into the *EditMask* field in the **Object Inspector**.

Note: For *TStringField* components, the *EditMask* property is also its display format.

To invoke the Input Mask editor for a field component

1. Select the component in the Fields editor or **Object Inspector**.
2. Click the Properties page in the **Object Inspector**.
3. Double-click the values column for the EditMask field in the **Object Inspector**, or click the ellipsis button. The Input Mask editor opens.

The Input Mask edit box lets you create and edit a mask format. The Sample Masks grid lets you select from predefined masks. If you select a sample mask, the mask format appears in the Input Mask edit box where you can modify it or use it as is. You can test the allowable user input for a mask in the Test Input edit box.

The Masks button enables you to load a custom set of masks—if you have created one—into the Sample Masks grid for easy selection.

See Also

Setting Display and Edit Properties at Design Time (see page 1860)

3.2.4 Programming with Delphi

Contains the Developer's Guide topics for programming with Delphi.

Topics

Name	Description
Building applications, components, and libraries (see page 1879)	
Creating international applications (see page 1920)	
Delphi programming fundamentals (see page 1935)	
Deploying applications (see page 1938)	
Developing the application user interface (see page 1955)	
Exception handling (see page 2014)	
Types of controls (see page 2031)	
Understanding the component library (see page 2053)	
Using the object model (see page 2062)	
Using the VCL/RTL (see page 2082)	
Working with components (see page 2142)	
Working with controls (see page 2149)	
Working with graphics and multimedia (see page 2169)	
Working with packages and components (see page 2208)	
Writing multi-threaded applications (see page 2224)	

3.2.4.1 Building applications, components, and libraries

Topics

Name	Description
Creating packages and DLLs (see page 1880)	
Creating Applications (see page 1883)	The most common types of applications you can design and build are: • GUI applications (see page 1902) • Console applications (see page 1900) • Service applications (see page 1903) • Packages and DLLs (see page 1882) GUI applications generally have an easy-to-use interface. Console applications run from a console window. Service applications are run as Windows services. These types of applications compile as executables with start-up code. You can create other types of projects such as packages and DLLs that result in creating packages or dynamically linkable libraries. These applications produce executable code without start-up code. Refer to Creating packages and DLLs (see page 1882).
Creating Web server applications (see page 1884)	
Enabling Help in applications (see page 1887)	
GUI applications (see page 1899)	
Service applications (see page 1902)	
Using data modules (see page 1909)	
Using the Object Repository (see page 1914)	
Writing applications using COM (see page 1918)	

Writing database applications (see page 1919)	

3.2.4.1.1 Creating packages and DLLs

Topics

Name	Description
Creating DLLs Containing VCL Components (C++) (see page 1880)	One of the strengths of DLLs is that a DLL created with one development tool can often be used by application written using a different development tool. When your DLL contains VCL components (such as forms) that are to be used by the calling application, you need to provide exported interface routines that use standard calling conventions, avoid C++ name mangling, and do not require the calling application to support the VCL libraries in order to work. To create VCL components that can be exported, use runtime packages. For example, suppose you want to create a DLL to display the... more (see page 1880)
Creating Packages and DLLs (see page 1882)	Dynamic link libraries (DLLs) are modules of compiled code that work in conjunction with an executable to provide functionality to an application. You can create DLLs in cross-platform programs. However, on Linux, DLLs (and packages) recompile as shared objects. DLLs and libraries should handle all exceptions to prevent the display of errors and warnings through Windows dialogs. The following compiler directives can be placed in library project files: ***Compiler directives for libraries***
Linking DLLs (see page 1883)	You can set the linker options for your DLL on the Linker page of the **Project Options** dialog box. The default check box on this page also creates an import library for your DLL. If compiling from the command line, invoke the linker, `ilink32.exe`, with the `-Tpd` switch. For example:
When to Use Packages and DLLs (see page 1883)	For most applications, packages provide greater flexibility and are easier to create than DLLs. However, there are several situations where DLLs would be better suited to your projects than packages: • Your code module will be called from non-Delphi applications. • You are extending the functionality of a Web server. • You are creating a code module to be used by third-party developers. • Your project is an OLE container. You cannot pass Delphi runtime type information (RTTI) across DLLs or from a DLL to an executable. If you pass an object from one DLL to another DLL or an executable, you will not... more (see page 1883)
Using DLLs in RAD Studio (C++) (see page 1883)	A Windows DLL can be used in a RAD Studio application just as it would be in any C++ application. To statically load a DLL when your C++ application is loaded, link the import library file for that DLL into your C++ application at link time. To add an import library to a C++ application, choose Project->Add to Project and select the `.lib` file you want to add to the libraries to be linked. The exported functions of that DLL then become available for use by your application. Prototype the DLL functions your application uses with the __declspec (dllimport) modifier:... more (see page 1883)

3.2.4.1.1.1 Creating DLLs Containing VCL Components (C++)

One of the strengths of DLLs is that a DLL created with one development tool can often be used by application written using a different development tool. When your DLL contains VCL components (such as forms) that are to be used by the calling application, you need to provide exported interface routines that use standard calling conventions, avoid C++ name mangling, and do not require the calling application to support the VCL libraries in order to work. To create VCL components that can be exported, use runtime packages.

For example, suppose you want to create a DLL to display the following simple dialog box:

The code for the dialog box DLL is as follows:

```
// DLLMAIN.H
//---------------------------------------------------------------------------
#ifndef dllMainH
#define dllMainH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------------------------------------------
class TYesNoDialog : public TForm
{
    __published:    // IDE-managed Components
        TLabel *LabelText;
        TButton *YesButton;
        TButton *NoButton;
        void __fastcall YesButtonClick(TObject *Sender);
        void __fastcall NoButtonClick(TObject *Sender);
    private:        // User declarations
        bool returnValue;
    public:         // User declarations
        virtual __fastcall TYesNoDialog(TComponent *Owner);
        bool __fastcall GetReturnValue();
};
// exported interface function
extern "C" __declspec(dllexport) bool InvokeYesNoDialog();
//---------------------------------------------------------------------------
extern TYesNoDialog *YesNoDialog;
//---------------------------------------------------------------------------
#endif
// DLLMAIN.CPP
//---------------------------------------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop
#include "dllMain.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TYesNoDialog *YesNoDialog;
//---------------------------------------------------------------------------
__fastcall TYesNoDialog::TYesNoDialog(TComponent *Owner)
    : TForm(Owner)
{
    returnValue = false;
}
//---------------------------------------------------------------------------
void __fastcall TYesNoDialog::YesButtonClick(TObject *Sender)
{
    returnValue = true;
    Close();
}
//---------------------------------------------------------------------------
void __fastcall TYesNoDialog::NoButtonClick(TObject *Sender)
{
    returnValue = false;
    Close();
}
//---------------------------------------------------------------------------
```

```
bool __fastcall TYesNoDialog::GetReturnValue()
{
    return returnValue;
}
//---------------------------------------------------------------------------
// exported standard C++ interface function that calls into VCL
bool InvokeYesNoDialog()
{
    bool returnValue;
    TYesNoDialog *YesNoDialog = new TYesNoDialog(NULL);
    YesNoDialog->ShowModal();
    returnValue = YesNoDialog->GetReturnValue();
    delete YesNoDialog;
    return returnValue;
}
//---------------------------------------------------------------------------
```

The code in this example displays the dialog and stores the value true in the private data member `returnValue` if the "Yes" button is pressed. Otherwise, `returnValue` is **false**. The public `GetReturnValue()` function retrieves the current value of `returnValue`.

To invoke the dialog and determine which button was pressed, the calling application calls the exported function `InvokeYesNoDialog()`. This function is declared in DLLMAIN.H as an exported function using C linkage (to avoid C++ name mangling) and the standard C calling convention. The function is defined in `DLLMAIN.CPP`.

By using a standard C function as the interface into the DLL, any calling application, whether or not it was created with RAD Studio, can use the DLL. The VCL functionality required to support the dialog is linked into the DLL itself, and the calling application does not need to know anything about it.

Note: When creating a DLL that uses the VCL, the required VCL components are linked into the DLL resulting in a certain amount of overhead. The impact of this overhead on the overall size of the application can be minimized by combining several components into one DLL that only needs one copy of the VCL support components.

3.2.4.1.1.2 Creating Packages and DLLs

Dynamic link libraries (DLLs) are modules of compiled code that work in conjunction with an executable to provide functionality to an application. You can create DLLs in cross-platform programs. However, on Linux, DLLs (and packages) recompile as shared objects.

DLLs and libraries should handle all exceptions to prevent the display of errors and warnings through Windows dialogs.

The following compiler directives can be placed in library project files:

Compiler directives for libraries

Compiler Directive	Description
{$LIBPREFIX 'string'}	Adds a specified prefix to the output file name. For example, you could specify {$LIBPREFIX 'dcl'} for a design-time package, or use {$LIBPREFIX''} to eliminate the prefix entirely.
{$LIBSUFFIX 'string'}	Adds a specified suffix to the output file name before the extension. For example, use {$LIBSUFFIX '-2.1.3'} in something.pas to generate something-2.1.3.bpl.
{$LIBVERSION 'string'}	Adds a second extension to the output file name after the .bpl extension. For example, use {$LIBVERSION '2.1.3'} in something.pas to generate something.bpl.2.1.3.

Packages are special DLLs used by Delphi applications, the IDE, or both. There are two kinds of packages: runtime packages and design-time packages. Runtime packages provide functionality to a program while that program is running. Design-time packages extend the functionality of the IDE.

For more information on packages, see Working with packages and components. (see page 2211)

See Also

Working with Packages and Components: Overview (see page 2211)

3.2.4.1.1.3 Linking DLLs

You can set the linker options for your DLL on the Linker page of the **Project Options** dialog box. The default check box on this page also creates an import library for your DLL. If compiling from the command line, invoke the linker, `ilink32.exe`, with the `-Tpd` switch. For example:

```
ilink32 /c /aa /Tpd c0d32.obj mydll.obj, mydll.dll, mydll.map, import32.lib cw32mt.lib
```

If you need an import library, use the -Gi switch to generate an import library.

You can optionally create an import library with the command line utility implib.exe. For more information on implib.exe, type `implib –h` at the command line.

3.2.4.1.1.4 When to Use Packages and DLLs

For most applications, packages provide greater flexibility and are easier to create than DLLs. However, there are several situations where DLLs would be better suited to your projects than packages:

- Your code module will be called from non-Delphi applications.
- You are extending the functionality of a Web server.
- You are creating a code module to be used by third-party developers.
- Your project is an OLE container.

You cannot pass Delphi runtime type information (RTTI) across DLLs or from a DLL to an executable. If you pass an object from one DLL to another DLL or an executable, you will not be able to use the **is** or **as** operators with the passed object. This is because the **is** and **as** operators need to compare RTTI. If you need to pass objects from a library, use packages instead, as these can share RTTI. Similarly, you should use packages instead of DLLs in Web Services because they are rely on Delphi RTTI.

3.2.4.1.1.5 Using DLLs in RAD Studio (C++)

A Windows DLL can be used in a RAD Studio application just as it would be in any C++ application.

To statically load a DLL when your C++ application is loaded, link the import library file for that DLL into your C++ application at link time. To add an import library to a C++ application, choose Project->Add to Project and select the `.lib` file you want to add to the libraries to be linked.

The exported functions of that DLL then become available for use by your application. Prototype the DLL functions your application uses with the __declspec (dllimport) modifier:

```
__declspec(dllimport) return_type imported_function_name(parameters);
```

To dynamically load a DLL during the run of a C++ application, include the import library, just as you would for static loading, and set the Delay load a DLL option for `ilink32`. You can also use the Windows API function `LoadLibrary()` to load the DLL, then use the API function `GetProcAddress()` to obtain pointers to the individual functions you want to use.

3.2.4.1.2 Creating Applications

The most common types of applications you can design and build are:

- GUI applications (see page 1902)

- Console applications (see page 1900)
- Service applications (see page 1903)
- Packages and DLLs (see page 1882)

GUI applications generally have an easy-to-use interface. Console applications run from a console window. Service applications are run as Windows services. These types of applications compile as executables with start-up code.

You can create other types of projects such as packages and DLLs that result in creating packages or dynamically linkable libraries. These applications produce executable code without start-up code. Refer to Creating packages and DLLs (see page 1882).

See Also

TForm

TScreen

TApplication

3.2.4.1.3 Creating Web server applications

Topics

Name	Description
Creating Web Services Applications (see page 1885)	Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services provide well-defined interfaces that describe the services provided. You use Web Services to produce or consume programmable services over the Internet using emerging standards such as XML, XML Schema, SOAP (Simple Object Access Protocol), and WSDL (Web Service Definition Language). Web Services use SOAP, a standard lightweight protocol for exchanging information in a distributed environment. It uses HTTP as a communications protocol and XML to encode remote procedure calls. You can build servers to implement Web Services... more (see page 1885)
Creating WebSnap Applications (see page 1885)	WebSnap provides a set of components and wizards for building advanced Web servers that interact with Web browsers. WebSnap components generate HTML or other MIME content for Web pages. WebSnap is for server-side development. To create a new WebSnap application, select File ▶ New ▶ Other and select the WebSnap tab in the New Items dialog box. Choose WebSnap Application. Then select the Web server application type (ISAPI/NSAPI, CGI, Apache). See the table in the topic Using Web Broker (see page 1885) for details. If you want to do client-side scripting instead of server-side scripting, you can use the InternetExpress technology. For more information on... more (see page 1885)
Creating Web Broker Applications (see page 1885)	You can use Web Broker to create Web server applications such as CGI applications or dynamic-link libraries (DLLs). These Web server applications can contain any nonvisual component. Components on the Internet category of the **Tool palette** enable you to create event handlers, programmatically construct HTML or XML documents, and transfer them to the client. To create a new Web server application using the Web Broker architecture, choose File ▶ New ▶ Other. In the New Items dialog box, select the Delphi Projects tab. Then select the New tab and double-click the Web Server Application. Then select the Web server application type:... more (see page 1885)
Creating Web Server Applications (see page 1886)	Web server applications are applications that run on servers that deliver Web content such as HTML Web pages or XML documents over the Internet. Examples of Web server applications include those which control access to a Web site, generate purchase orders, or respond to information requests. You can create several different types of Web server applications using the following technologies: • Web Broker (see page 1885) • WebSnap (see page 1885) • IntraWeb (see page 2259) • Web Services (see page 1885)

3.2.4.1.3.1 Creating Web Services Applications

Web Services are self-contained modular applications that can be published and invoked over a network (such as the World Wide Web). Web Services provide well-defined interfaces that describe the services provided. You use Web Services to produce or consume programmable services over the Internet using emerging standards such as XML, XML Schema, SOAP (Simple Object Access Protocol), and WSDL (Web Service Definition Language).

Web Services use SOAP, a standard lightweight protocol for exchanging information in a distributed environment. It uses HTTP as a communications protocol and XML to encode remote procedure calls.

You can build servers to implement Web Services and clients that call on those services. You can write clients for arbitrary servers to implement Web Services that respond to SOAP messages, and servers to publish Web Services for use by arbitrary clients.

Refer to Using Web Services (see page 2291) for more information on Web Services.

See Also

Creating Web Broker Applications (see page 1885)

Creating WebSnap Applications (see page 1885)

3.2.4.1.3.2 Creating WebSnap Applications

WebSnap provides a set of components and wizards for building advanced Web servers that interact with Web browsers. WebSnap components generate HTML or other MIME content for Web pages. WebSnap is for server-side development.

To create a new WebSnap application, select **File** ▶ **New** ▶ **Other** and select the WebSnap tab in the New Items dialog box. Choose WebSnap Application. Then select the Web server application type (ISAPI/NSAPI, CGI, Apache). See the table in the topic Using Web Broker (see page 1885) for details.

If you want to do client-side scripting instead of server-side scripting, you can use the InternetExpress technology. For more information on InternetExpress, see Building Web applications using InternetExpress (see page 1522).

For more information on WebSnap, see Creating Internet Server Applications. (see page 2251)

See Also

Creating Web Broker Applications (see page 1885)

Creating Web Services Applications (see page 1885)

3.2.4.1.3.3 Creating Web Broker Applications

You can use Web Broker to create Web server applications such as CGI applications or dynamic-link libraries (DLLs). These Web server applications can contain any nonvisual component. Components on the Internet category of the **Tool palette** enable you to create event handlers, programmatically construct HTML or XML documents, and transfer them to the client.

To create a new Web server application using the Web Broker architecture, choose **File** ▶ **New** ▶ **Other**. In the New Items dialog box, select the Delphi Projects tab. Then select the New tab and double-click the Web Server Application. Then select the Web server application type:

Web server applications

Web server application type	Description
ISAPI and NSAPI Dynamic Link Library	ISAPI and NSAPI Web server applications are DLLs that are loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by TISAPIApplication. Each request message is handled in a separate execution thread. Selecting this type of application adds the library header of the project files and required entries to the uses list and exports clause of the project file.
CGI Stand-alone executable	CGI Web server applications are console applications that receive requests from clients on standard input, process those requests, and sends back the results to the server on standard output to be sent to the client. Selecting this type of application adds the required entries to the uses clause of the project file and adds the appropriate $APPTYPE directive to the source.
Apache Shared Module (DLL)	Selecting this type of application sets up your project as a DLL. Apache Web server applications are DLLs loaded by the Web server. Information is passed to the DLL, processed, and returned to the client by the Web server.
Web App Debugger stand-alone executable	Selecting this type of application sets up an environment for developing and testing Web server applications. Web App Debugger applications are executable files loaded by the Web server. This type of application is not intended for deployment.

CGI applications use more system resources on the server, so complex applications are better created as ISAPI, NSAPI, or Apache DLL applications. When writing cross-platform applications, you should select CGI stand-alone or Apache Shared Module (DLL) for Web server development. These are also the same options you see when creating WebSnap and Web Service applications.

For more information on building Web server applications, see Creating Internet Server Applications. (see page 2251)

See Also

Creating WebSnap Applications (see page 1885)

Creating Web Services Applications (see page 1885)

3.2.4.1.3.4 Creating Web Server Applications

Web server applications are applications that run on servers that deliver Web content such as HTML Web pages or XML documents over the Internet. Examples of Web server applications include those which control access to a Web site, generate purchase orders, or respond to information requests.

You can create several different types of Web server applications using the following technologies:

- Web Broker (see page 1885)
- WebSnap (see page 1885)
- IntraWeb (see page 2259)
- Web Services (see page 1885)

See Also

Creating Web Broker Applications (see page 1885)

Creating WebSnap Applications (see page 1885)

Creating Web Services Applications (see page 1885)

3.2.4.1.4 Enabling Help in applications

Topics

Name	Description
Asking the Help Manager for Information (see page 1890)	Help viewers communicate with the Help Manager through the *IHelpManager* interface, an instance of which is returned to them when they register with the Help Manager. *IHelpManager* allows the Help viewer to communicate four things: • A request for the window handle of the currently active control. • A request for the name of the Help file which the Help Manager believes should contain help for the currently active control. • A request for the path to that Help file. • A notification that the Help viewer is shutting itself down in response to something other than a request from the Help Manager that... more (see page 1890)
Calling a Help System Directly (see page 1890)	For additional Help system functionality not provided by VCL applications, *TApplication* provides a read-only property that allows direct access to the Help system. This property is an instance of an implementation of the interface *IHelpSystem*. *IHelpSystem* and *IHelpManager* are implemented by the same object, but one interface is used to allow the application to talk to the Help Manager, and one is used to allow the Help viewers to talk to the Help Manager.
Communicating with the Help Manager (see page 1890)	The ICustomHelpViewer provides four functions that can be used to communicate system information with the Help Manager: • *GetViewerName* • *NotifyID* • *ShutDown* • SoftShutDown The Help Manager calls through these functions in the following circumstances: • *ICustomHelpViewer.GetViewerName* : *String* is called when the Help Manager wants to know the name of the viewer (for example, if the application is asked to display a list of all registered viewers). This information is returned via a string, and is required to be logically static (that is, it cannot change during the operation of the application). Multibyte character sets are not supported. • *ICustomHelpViewer.NotifyID(const ViewerID: Integer)* is called... more (see page 1890)
Customizing the IDE Help System (see page 1891)	The IDE supports multiple Help viewers in exactly the same way that a VCL application does: it delegates Help requests to the Help Manager, which forwards them to registered Help viewers. The IDE makes use of the same WinHelpViewer that the VCL uses. The IDE comes with two Help viewers installed: the HyperHelp viewer, which allows Help requests to be forwarded to HyperHelp, an external WinHelp emulator under which the Kylix Help files are viewed, and the Man page viewer, which allows you to access the Man system installed on most Unix machines. Because it is necessary for Kylix... more (see page 1891)

Displaying Keyword-based Help (see page 1892)	Help requests typically come through to the Help viewer as either *keyword-based* Help, in which case the viewer is asked to provide help based upon a particular string, or as *context-based* Help, in which case the viewer is asked to provide help based upon a particular numeric identifier. **Note:** Numeric Help contexts are the default form of Help requests in applications running under Windows, which use the WinHelp system. *ICustomHelpViewer* implementations are required to provide support for keyword-based Help requests, while *IExtendedHelpViewer* implementations are required to support context-based Help requests. *ICustomHelpViewer* provides three methods for handling keyword-based Help: • *UnderstandsKeyword* • *GetHelpStrings... more (see page 1892)*
Displaying Tables of Contents (see page 1893)	*ICustomHelpViewer* provides two methods relating to displaying tables of contents: • *CanShowTableOfContents* • *ShowTableOfContents* The theory behind their operation is similar to the operation of the keyword Help request functions: the Help Manager first queries all Help viewers by calling *ICustomHelpViewer.CanShowTableOfContents : Boolean* and then invokes a particular Help viewer by calling *ICustomHelpViewer.ShowTableOfContents*. It is reasonable for a particular viewer to refuse to allow requests to support a table of contents. The Man page viewer does this, for example, because the concept of a table of contents does not map well to the way Man pages work; the HyperHelp viewer supports... more (see page 1893)
Enabling Help in Applications (see page 1893)	VCL applications support displaying Help using an object-based mechanism that allows Help requests to be passed on to one of multiple external Help viewers. To support this, an application must include a class that implements the *ICustomHelpViewer* interface (and, optionally, one of several interfaces descended from it), and registers itself with the global Help Manager. VCL applications provide an instance of *TWinHelpViewer*, which implements all of these interfaces and provides a link between applications and WinHelp. The Help Manager maintains a list of registered viewers and passes requests to them in a two-phase process: it first asks each viewer... more (see page 1893)
Help System Interfaces (see page 1894)	The Help system allows communication between your application and Help viewers through a series of interfaces. These interfaces are all defined in the HelpIntfs.pas, which also contains the implementation of the Help Manager. ICustomHelpViewer provides support for displaying Help based upon a provided keyword and for displaying a table of contents listing all Help available in a particular viewer. IExtendedHelpViewer provides support for displaying Help based upon a numeric Help context and for displaying topics; in most Help systems, topics function as high-level keywords (for example, "IntToStr" might be a keyword in the Help system, but "String manipulation routines" could... more (see page 1894)
How TApplication Processes VCL Help (see page 1895)	*TApplication* in the VCL provides four methods that are accessible from application code: ***Help methods in TApplication***
How VCL Controls Process Help (see page 1895)	All VCL controls that derive from *TControl* expose several properties that are used by the Help system: *HelpType*, *HelpContext*, and *HelpKeyword*. The *HelpType* property contains an instance of an enumerated type that determines if the control's designer expects help to be provided via keyword-based Help or context-based Help. If the *HelpType* is set to *htKeyword*, then the Help system expects the control to use keyword-based Help, and the Help system only looks at the contents of the *HelpKeyword* property. Conversely, if the *HelpType* is set to *htContext*, the Help system expects the control to use... more (see page 1895)

Implementing ICustomHelpViewer (see page 1895)	The ICustomHelpViewer interface contains three types of methods: methods used to communicate system-level information (for example, information not related to a particular Help request) with the Help Manager; methods related to showing Help based upon a keyword provided by the Help Manager; and methods for displaying a table of contents. For information on ICustomHelpViewer methods, see • Communicating with the Help Manager (see page 1890) • Displaying keyword-based Help (see page 1892) • Asking the Help Manager for information (see page 1890)
Implementing IExtendedHelpViewer (see page 1896)	*ICustomHelpViewer* only provides direct support for keyword-based Help. Some Help systems (especially WinHelp) work by associating numbers (known as *context IDs*) with keywords in a fashion which is internal to the Help system and therefore not visible to the application. Such systems require that the application support context-based Help in which the application invokes the Help system with that context, rather than with a string, and the Help system translates the number itself. Applications can talk to systems requiring context-based Help by extending the object that implements *ICustomHelpViewer* to also implement *IExtendedHelpViewer*. *IExtendedHelpViewer* also provides support for talking... more (see page 1896)
Implementing IHelpSelector (see page 1897)	*IHelpSelector* is a companion to *ICustomHelpViewer*. When more than one registered viewer claims to provide support for a given keyword, context, or topic, or provides a table of contents, the Help Manager must choose between them. In the case of contexts or topics, the Help Manager *always* selects the first Help viewer that claims to provide support. In the case of keywords or the table of context, the Help Manager will, by default, select the first Help viewer. This behavior can be overridden by an application. To override the decision of the Help Manager in such cases, an application... more (see page 1897)
Registering Help System Objects (see page 1897)	For the Help Manager to communicate with them, objects that implement *ICustomHelpViewer, IExtendedHelpViewer, ISpecialWinHelpViewer,* and *IHelpSelector* must register with the Help Manager. To register Help system objects with the Help Manager, you need to: • Register the Help viewer. • Register the Help Selector.
Using Help in a VCL Application (see page 1898)	The following sections explain how to use Help within a VCL application. • How TApplication Processes VCL Help (see page 1895) • How VCL controls process Help (see page 1895) • Calling a Help system directly (see page 1890) • Using IHelpSystem (see page 1898)
Using IHelpSystem (see page 1898)	*IHelpSystem* allows an application to do three things: • Provides path information to the Help Manager. • Provides a new Help selector. • Asks the Help Manager to display Help. Providing path information is important because the Help Manager is platform-independent and Help system-independent and so is not able to ascertain the location of Help files. If an application expects Help to be provided by an external Help system that is not able to ascertain file locations itself, it must provide this information through the *IHelpSystem's ProvideHelpPath* method, which allows the information to become available through the *IHelpManager's GetHelpPath* method. (This information propagates... more (see page 1898)

3.2.4.1.4.1 Asking the Help Manager for Information

Help viewers communicate with the Help Manager through the *IHelpManager* interface, an instance of which is returned to them when they register with the Help Manager. *IHelpManager* allows the Help viewer to communicate four things:

- A request for the window handle of the currently active control.
- A request for the name of the Help file which the Help Manager believes should contain help for the currently active control.
- A request for the path to that Help file.
- A notification that the Help viewer is shutting itself down in response to something other than a request from the Help Manager that it do so.

IHelpManager.GetHandle : *LongInt* is called by the Help viewer if it needs to know the handle of the currently active control; the result is a window handle.

IHelpManager.GetHelpFile: *String* is called by the Help viewer if it needs to know the name of the Help file which the currently active control believes contains its Help.

IHelpManager.Release is called to notify the Help Manager when a Help viewer is disconnecting. It should never be called in response to a request through *ICustomHelpViewer.ShutDown*; it is only used to notify the Help Manager of unexpected disconnects.

See Also

Enabling Help in Applications (see page 1893)

Help System Interfaces (see page 1894)

Communicating with the Help Manager (see page 1890)

Implementing ICustomHelpViewer (see page 1895)

Displaying Tables of Contents (see page 1893)

Implementing IExtendedHelpViewer (see page 1896)

Implementing IHelpSelector (see page 1897)

Registering Help System Objects (see page 1897)

3.2.4.1.4.2 Calling a Help System Directly

For additional Help system functionality not provided by VCL applications, *TApplication* provides a read-only property that allows direct access to the Help system. This property is an instance of an implementation of the interface *IHelpSystem*. *IHelpSystem* and *IHelpManager* are implemented by the same object, but one interface is used to allow the application to talk to the Help Manager, and one is used to allow the Help viewers to talk to the Help Manager.

See Also

Enabling Help in Applications (see page 1893)

How TApplication Processes VCL Help (see page 1895)

3.2.4.1.4.3 Communicating with the Help Manager

The ICustomHelpViewer provides four functions that can be used to communicate system information with the Help Manager:

- *GetViewerName*
- *NotifyID*
- *ShutDown*
- SoftShutDown

The Help Manager calls through these functions in the following circumstances:

- *ICustomHelpViewer.GetViewerName* : *String* is called when the Help Manager wants to know the name of the viewer (for example, if the application is asked to display a list of all registered viewers). This information is returned via a string, and is required to be logically static (that is, it cannot change during the operation of the application). Multibyte character sets are not supported.
- *ICustomHelpViewer.NotifyID(const ViewerID: Integer)* is called **immediately** following registration to provide the viewer with a unique cookie that identifies it. This information must be stored off for later use; if the viewer shuts down on its own (as opposed to in response to a notification from the Help Manager), it must provide the Help Manager with the identifying cookie so that the Help Manager can release all references to the viewer. (Failing to provide the cookie, or providing the wrong one, causes the Help Manager to potentially release references to the wrong viewer.)
- *ICustomHelpViewer.ShutDown* is called by the Help Manager to notify the Help viewer that the Manager is shutting down and that any resources the Help viewer has allocated should be freed. It is recommended that all resource freeing be delegated to this method.
- *ICustomHelpViewer.SoftShutDown* is called by the Help Manager to ask the Help viewer to close any externally visible manifestations of the Help system (for example, windows displaying Help information) without unloading the viewer.

See Also

Enabling Help in Applications (see page 1893)

Help System Interfaces (see page 1894)

Implementing ICustomHelpViewer (see page 1895)

Asking the Help Manager for Information (see page 1890)

Displaying Tables of Contents (see page 1893)

Implementing IExtendedHelpViewer (see page 1896)

Implementing IHelpSelector (see page 1897)

Registering Help System Objects (see page 1897)

3.2.4.1.4.4 Customizing the IDE Help System

The IDE supports multiple Help viewers in exactly the same way that a VCL application does: it delegates Help requests to the Help Manager, which forwards them to registered Help viewers. The IDE makes use of the same WinHelpViewer that the VCL uses.

The IDE comes with two Help viewers installed: the HyperHelp viewer, which allows Help requests to be forwarded to HyperHelp, an external WinHelp emulator under which the Kylix Help files are viewed, and the Man page viewer, which allows you to access the Man system installed on most Unix machines. Because it is necessary for Kylix Help to work, the HyperHelp viewer may not be removed; the Man page viewer ships in a separate package whose source is available in the examples directory.

To install a new Help viewer in the IDE, you do exactly what you would do in a VCL application, with one difference. You write an object that implements *ICustomHelpViewer* (and, if desired, *IExtendedHelpViewer*) to forward Help requests to the external viewer of your choice, and you register the *ICustomHelpViewer* with the IDE.

To register a custom Help viewer with the IDE:

1. Make sure that the unit implementing the Help viewer contains HelpIntfs.pas.
2. Build the unit into a design-time package registered with the IDE, and build the package with runtime packages turned on. (This is necessary to ensure that the Help Manager instance used by the unit is the same as the Help Manager instance used by the IDE.)
3. Make sure that the Help viewer exists as a global instance within the unit.
4. In the initialization section of the unit, make sure that the instance is passed to the *RegisterHelpViewer* function.

See Also

Enabling Help in Applications (see page 1893)

How TApplication Processes VCL Help (see page 1895)

3.2.4.1.4.5 Displaying Keyword-based Help

Help requests typically come through to the Help viewer as either *keyword-based* Help, in which case the viewer is asked to provide help based upon a particular string, or as *context-based* Help, in which case the viewer is asked to provide help based upon a particular numeric identifier.

Note: Numeric Help contexts are the default form of Help requests in applications running under Windows, which use the WinHelp system.

ICustomHelpViewer implementations are required to provide support for keyword-based Help requests, while *IExtendedHelpViewer* implementations are required to support context-based Help requests.

ICustomHelpViewer provides three methods for handling keyword-based Help:

- *UnderstandsKeyword*
- *GetHelpStrings*
- *ShowHelp*

```
ICustomHelpViewer.UnderstandsKeyword(const HelpString: String): Integer
int__fastcall ICustomHelpViewer::UnderstandsKeyword(const AnsiString HelpString)
```

is the first of the three methods called by the Help Manager, which will call each registered Help viewer with the same string to ask if the viewer provides help for that string; the viewer is expected to respond with an integer indicating how many different Help pages it can display in response to that Help request. The viewer can use any method it wants to determine this—inside the IDE, the HyperHelp viewer maintains its own index and searches it. If the viewer does not support help on this keyword, it should return zero. Negative numbers are currently interpreted as meaning zero, but this behavior is not guaranteed in future releases.

```
ICustomHelpViewer.GetHelpStrings(const HelpString: String): TStringList
Classes::TStringList*__fastcall ICustomHelpViewer::GetHelpStrings(const AnsiString HelpString)
```

is called by the Help Manager if more than one viewer can provide Help on a topic. The viewer is expected to return a *TStringList*, which is freed by the Help Manager. The strings in the returned list should map to the pages available for that keyword, but the characteristics of that mapping can be determined by the viewer. In the case of the WinHelp viewer on Windows and the HyperHelp viewer on Linux, the string list always contains exactly one entry. HyperHelp provides its own indexing, and duplicating that elsewhere would be pointless duplication. In the case of the Man page viewer (Linux), the string list consists of multiple strings, one for each section of the manual which contains a page for that keyword.

```
ICustomHelpViewer.ShowHelp(const HelpString: String)
void__fastcall ICustomHelpViewer::ShowHelp(const AnsiString HelpString)
```

is called by the Help Manager if it needs the Help viewer to display help for a particular keyword. This is the last method call in the operation; it is guaranteed to never be called unless the *UnderstandsKeyword* method is invoked first.

See Also

Enabling Help in Applications (see page 1893)

Help System Interfaces (see page 1894)

Communicating with the Help Manager (see page 1890)

Implementing ICustomHelpViewer (see page 1895)

Asking the Help Manager for Information (see page 1890)

Displaying Tables of Contents (see page 1893)

Implementing IExtendedHelpViewer (see page 1896)

Implementing IHelpSelector (see page 1897)

Registering Help System Objects (see page 1897)

3.2.4.1.4.6 Displaying Tables of Contents

ICustomHelpViewer provides two methods relating to displaying tables of contents:

- *CanShowTableOfContents*
- *ShowTableOfContents*

The theory behind their operation is similar to the operation of the keyword Help request functions: the Help Manager first queries all Help viewers by calling *ICustomHelpViewer.CanShowTableOfContents : Boolean* and then invokes a particular Help viewer by calling *ICustomHelpViewer.ShowTableOfContents*.

It is reasonable for a particular viewer to refuse to allow requests to support a table of contents. The Man page viewer does this, for example, because the concept of a table of contents does not map well to the way Man pages work; the HyperHelp viewer supports a table of contents, on the other hand, by passing the request to display a table of contents directly to WinHelp on Windows and HyperHelp on Linux. It is not reasonable, however, for an implementation of *ICustomHelpViewer* to respond to queries through *CanShowTableOfContents* with the answer *True*, and then ignore requests through *ShowTableOfContents*.

See Also

Enabling Help in Applications (see page 1893)

Help System Interfaces (see page 1894)

Communicating with the Help Manager (see page 1890)

Implementing ICustomHelpViewer (see page 1895)

Asking the Help Manager for Information (see page 1890)

Displaying Tables of Contents

Implementing IExtendedHelpViewer (see page 1896)

Implementing IHelpSelector (see page 1897)

Registering Help System Objects (see page 1897)

3.2.4.1.4.7 Enabling Help in Applications

VCL applications support displaying Help using an object-based mechanism that allows Help requests to be passed on to one of multiple external Help viewers. To support this, an application must include a class that implements the *ICustomHelpViewer* interface (and, optionally, one of several interfaces descended from it), and registers itself with the global Help Manager.

VCL applications provide an instance of *TWinHelpViewer*, which implements all of these interfaces and provides a link between applications and WinHelp.

The Help Manager maintains a list of registered viewers and passes requests to them in a two-phase process: it first asks each viewer if it can provide support for a particular Help keyword or context, and then it passes the Help request on to the viewer which says it can provide such support.

If more than one viewer supports the keyword, as would be the case in an application that had registered viewers for both WinHelp and HyperHelp on Windows, the Help Manager can display a selection box through which the user of the application can determine which Help viewer to invoke. Otherwise, it displays the first responding Help system encountered.

See Also

Help System Interfaces (see page 1894)

Communicating with the Help Manager (see page 1890)

Implementing ICustomHelpViewer (see page 1895)

Asking the Help Manager for Information (see page 1890)

Displaying Tables of Contents (see page 1893)

Implementing IExtendedHelpViewer (see page 1896)

Implementing IHelpSelector (see page 1897)

Registering Help System Objects (see page 1897)

How TApplication Processes VCL Help (see page 1895)

Calling a Help System Directly (see page 1890)

Customizing the IDE Help System (see page 1891)

3.2.4.1.4.8 Help System Interfaces

The Help system allows communication between your application and Help viewers through a series of interfaces. These interfaces are all defined in the HelpIntfs.pas, which also contains the implementation of the Help Manager.

ICustomHelpViewer provides support for displaying Help based upon a provided keyword and for displaying a table of contents listing all Help available in a particular viewer.

IExtendedHelpViewer provides support for displaying Help based upon a numeric Help context and for displaying topics; in most Help systems, topics function as high-level keywords (for example, "IntToStr" might be a keyword in the Help system, but "String manipulation routines" could be the name of a topic).

ISpecialWinHelpViewer provides support for responding to specialized WinHelp messages that an application running under Windows may receive and which are not easily generalizable. In general, only applications operating in the Windows environment need to implement this interface, and even then it is only required for applications that make extensive use of non-standard WinHelp messages.

IHelpManager provides a mechanism for the Help viewer to communicate back to the application's Help Manager and request additional information. *IHelpManager* is obtained at the time the Help viewer registers itself.

IHelpSystem provides a mechanism through which *TApplication* passes Help requests on to the Help system. *TApplication* obtains an instance of an object which implements both *IHelpSystem* and *IHelpManager* at application load time and exports that instance as a property; this allows other code within the application to file Help requests directly when appropriate.

IHelpSelector provides a mechanism through which the Help system can invoke the user interface to ask which Help viewer should be used in cases where more than one viewer is capable of handling a Help request, and to display a Table of Contents. This display capability is not built into the Help Manager directly to allow the Help Manager code to be identical regardless of which widget set or class library is in use.

See Also

Enabling Help in Applications (see page 1893)

Communicating with the Help Manager (see page 1890)

Implementing ICustomHelpViewer (see page 1895)

Asking the Help Manager for Information (see page 1890)

Displaying Tables of Contents (see page 1893)

Implementing IExtendedHelpViewer (see page 1896)

Implementing IHelpSelector (see page 1897)

Registering Help System Objects (see page 1897)

3.2.4.1.4.9 How TApplication Processes VCL Help

TApplication in the VCL provides four methods that are accessible from application code:

Help methods in TApplication

Method	Description
HelpCommand	Takes a Windows Help style HELP_COMMAND and passes it off to WinHelp. Help requests forwarded through this mechanism are passed only to implementations of IspecialWinHelpViewer.
HelpContext	Invokes the Help System with a request for context-based Help.
HelpKeyword	Invokes the HelpSystem with a request for keyword-based Help.
HelpJump	Requests the display of a particular topic.

All four functions take the data passed to them and forward it through a data member of *TApplication*, which represents the Help system. That data member is directly accessible through the property *HelpSystem*.

See Also

Enabling Help in Applications (see page 1893)

3.2.4.1.4.10 How VCL Controls Process Help

All VCL controls that derive from *TControl* expose several properties that are used by the Help system: *HelpType*, *HelpContext*, and *HelpKeyword*.

The *HelpType* property contains an instance of an enumerated type that determines if the control's designer expects help to be provided via keyword-based Help or context-based Help. If the *HelpType* is set to *htKeyword*, then the Help system expects the control to use keyword-based Help, and the Help system only looks at the contents of the *HelpKeyword* property. Conversely, if the *HelpType* is set to *htContext*, the Help system expects the control to use context-based Help and only looks at the contents of the *HelpContext* property.

In addition to the properties, controls expose a single method, *InvokeHelp*, that can be called to pass a request to the Help system. It takes no parameters and calls the methods in the global Application object, which correspond to the type of Help the control supports.

Help messages are automatically invoked when F1 is pressed because the *KeyDown* method of *TWinControl* calls *InvokeHelp*.

See Also

How TApplication Processes VCL Help (see page 1895)

3.2.4.1.4.11 Implementing ICustomHelpViewer

The ICustomHelpViewer interface contains three types of methods: methods used to communicate system-level information (for example, information not related to a particular Help request) with the Help Manager; methods related to showing Help based upon a keyword provided by the Help Manager; and methods for displaying a table of contents.

For information on ICustomHelpViewer methods, see

- Communicating with the Help Manager (see page 1890)
- Displaying keyword-based Help (see page 1892)
- Asking the Help Manager for information (see page 1890)

See Also

Enabling Help in Applications (see page 1893)

Help System Interfaces (see page 1894)

Communicating with the Help Manager (see page 1890)

Asking the Help Manager for Information (see page 1890)

Displaying Tables of Contents (see page 1893)

Implementing IExtendedHelpViewer (see page 1896)

Implementing IHelpSelector (see page 1897)

Registering Help System Objects (see page 1897)

3.2.4.1.4.12 Implementing IExtendedHelpViewer

ICustomHelpViewer only provides direct support for keyword-based Help. Some Help systems (especially WinHelp) work by associating numbers (known as *context IDs*) with keywords in a fashion which is internal to the Help system and therefore not visible to the application. Such systems require that the application support context-based Help in which the application invokes the Help system with that context, rather than with a string, and the Help system translates the number itself.

Applications can talk to systems requiring context-based Help by extending the object that implements *ICustomHelpViewer* to also implement *IExtendedHelpViewer*. *IExtendedHelpViewer* also provides support for talking to Help systems that allow you to jump directly to high-level topics instead of using keyword searches. The built-in WinHelp viewer does this for you automatically.

IExtendedHelpViewer exposes four functions. Two of them—*UnderstandsContext* and *DisplayHelpByContext*—are used to support context-based Help; the other two—*UnderstandsTopic* and *DisplayTopic*—are used to support topics.

When an application user presses F1, the Help Manager calls

```
IExtendedHelpViewer.UnderstandsContext(const ContextID: Integer;
const HelpFileName: String): Boolean
int__fastcall IExtendedHelpViewer::UnderstandsContext(const int ContextID, AnsiString
HelpFileName)
```

and the currently activated control supports context-based, rather than keyword-based Help. As with *ICustomHelpViewer.UnderstandsKeyword*, the Help Manager queries all registered Help viewers iteratively. Unlike the case with *ICustomHelpViewer.UnderstandsKeyword*, however, if more than one viewer supports a specified context, the first registered viewer with support for a given context is invoked.

The Help Manager calls

```
IExtendedHelpViewer.DisplayHelpByContext(const ContextID: Integer;
const HelpFileName: String)
void__fastcall IExtendedHelpViewer::DisplayHelpByContext(const int ContextID, AnsiString
HelpFileName)
```

after it has polled the registered Help viewers.

The topic support functions work the same way:

```
IExtendedHelpViewer.UnderstandsTopic(const Topic: String): Boolean
bool__fastcall IExtendedHelpViewer::UnderstandsTopic(const AnsiString Topic)
```

is used to poll the Help viewers asking if they support a topic;

```
IExtendedHelpViewer.DisplayTopic(const Topic: String)
void__fastcall IExtendedHelpViewer::DisplayTopic(const AnsiString Topic)
```

is used to invoke the first registered viewer which reports that it is able to provide help for that topic.

See Also

Enabling Help in Applications (see page 1893)

Help System Interfaces (see page 1894)

Communicating with the Help Manager (see page 1890)

Implementing ICustomHelpViewer (see page 1895)

Asking the Help Manager for Information (see page 1890)

Displaying Tables of Contents (see page 1893)

Implementing IExtendedHelpViewer

Implementing IHelpSelector (see page 1897)

Registering Help System Objects (see page 1897)

3.2.4.1.4.13 Implementing IHelpSelector

IHelpSelector is a companion to *ICustomHelpViewer*. When more than one registered viewer claims to provide support for a given keyword, context, or topic, or provides a table of contents, the Help Manager must choose between them. In the case of contexts or topics, the Help Manager *always* selects the first Help viewer that claims to provide support. In the case of keywords or the table of context, the Help Manager will, by default, select the first Help viewer. This behavior can be overridden by an application.

To override the decision of the Help Manager in such cases, an application must register a class that provides an implementation of the *IHelpSelector* interface. *IHelpSelector* exports two functions: *SelectKeyword*, and *TableOfContents*. Both take as arguments a TStrings containing, one by one, either the possible keyword matches or the names of the viewers claiming to provide a table of contents. The implementor is required to return the index (in the *TStringList*) that represents the selected string; the *TStringList* is then freed by the Help Manager.

Note: The Help Manager may get confused if the strings are rearranged; it is recommended that implementors of *IHelpSelector* refrain from doing this. The Help system only supports *one* HelpSelector; when new selectors are registered, any previously existing selectors are disconnected.

See Also

Enabling Help in Applications (see page 1893)

Help System Interfaces (see page 1894)

Communicating with the Help Manager (see page 1890)

Implementing ICustomHelpViewer (see page 1895)

Asking the Help Manager for Information (see page 1890)

Displaying Keyword-based Help (see page 1892)

Displaying Tables of Contents (see page 1893)

Implementing IExtendedHelpViewer (see page 1896)

Registering Help System Objects (see page 1897)

3.2.4.1.4.14 Registering Help System Objects

For the Help Manager to communicate with them, objects that implement *ICustomHelpViewer, IExtendedHelpViewer, ISpecialWinHelpViewer,* and *IHelpSelector* must register with the Help Manager.

To register Help system objects with the Help Manager, you need to:

- Register the Help viewer.
- Register the Help Selector.

Registering Help viewers

The unit that contains the object implementation must use HelpIntfs. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must assign the instance variable and pass it to the function *RegisterViewer*. *RegisterViewer* is a flat function exported by the HelpIntfs unit, which takes as an argument an *ICustomHelpViewer* and returns an *IHelpManager*. The *IHelpManager* should be stored for future use.

Registering Help selectors

The unit that contains the object implementation must use Forms in the VCL. An instance of the object must be declared in the **var** section of the implementing unit.

The initialization section of the implementing unit must register the Help selector through the *HelpSystem* property of the global Application object:

```
Application.HelpSystem.AssignHelpSelector(myHelpSelectorInstance)
Application->HelpSystem->AssignHelpSelector(myHelpSelectorInstance)
```

This procedure does not return a value.

See Also

Enabling Help in Applications (see page 1893)

Help System Interfaces (see page 1894)

Communicating with the Help Manager (see page 1890)

Implementing ICustomHelpViewer (see page 1895)

Asking the Help Manager for Information (see page 1890)

Displaying Keyword-based Help (see page 1892)

Displaying Tables of Contents (see page 1893)

Implementing IExtendedHelpViewer (see page 1896)

Implementing IHelpSelector (see page 1897)

3.2.4.1.4.15 Using Help in a VCL Application

The following sections explain how to use Help within a VCL application.

- How TApplication Processes VCL Help (see page 1895)
- How VCL controls process Help (see page 1895)
- Calling a Help system directly (see page 1890)
- Using IHelpSystem (see page 1898)

See Also

Enabling Help in Applications (see page 1893)

3.2.4.1.4.16 Using IHelpSystem

IHelpSystem allows an application to do three things:

- Provides path information to the Help Manager.
- Provides a new Help selector.
- Asks the Help Manager to display Help.

Providing path information is important because the Help Manager is platform-independent and Help system-independent and so is not able to ascertain the location of Help files. If an application expects Help to be provided by an external Help system that is not able to ascertain file locations itself, it must provide this information through the *IHelpSystem's ProvideHelpPath* method, which allows the information to become available through the *IHelpManager's GetHelpPath* method. (This information propagates outward only if the Help viewer asks for it.)

Assigning a Help selector allows the Help Manager to delegate decision-making in cases where multiple external Help systems can provide Help for the same keyword. For more information, see the topic Implementing IHelpSelector (see page 1897).

IHelpSystem exports four procedures and one function to request the Help Manager to display Help:

- *ShowHelp*
- *ShowContextHelp*
- *ShowTopicHelp*
- *ShowTableOfContents*
- *Hook*

Hook is intended entirely for WinHelp compatibility; it allows processing of WM_HELP messages that cannot be mapped directly onto requests for keyword-based, context-based, or topic-based Help. The other methods each take two arguments: the keyword, context ID, or topic for which help is being requested, and the Help file in which it is expected that help can be found.

In general, unless you are asking for topic-based help, it is equally effective and more clear to pass help requests to the Help Manager through the *InvokeHelp* method of your control.

See Also

Enabling Help in Applications (see page 1893)

How TApplication Processes VCL Help (see page 1895)

Calling a Help System Directly (see page 1890)

3.2.4.1.5 GUI applications

Topics

Name	Description
Setting IDE, Project, and Compiler Options (see page 1900)	In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE. To specify various options for your project, choose **Project** ▶ **Options**.
Console Applications (see page 1900)	Console applications are 32-bit programs that run without a graphical interface, in a console window. These applications typically don't require much user input and perform a limited set of functions. Any application that contains:
MDI Applications (see page 1901)	
SDI Applications (see page 1901)	
User Interface Models (see page 1902)	Any form can be implemented as a single document interface (SDI) or multiple document interface (MDI) form. An SDI application (see page 1901) normally contains a single document view. In an MDI application, more than one document or child window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors. For more information on developing the UI for an application, see Developing the application user interface (see page 1983).

GUI Applications (see page 1902)	A graphical user interface (GUI) application is one that is designed using graphical features such as windows, menus, dialog boxes, and features that make the application easy to use. When you compile a GUI application, an executable file with start-up code is created. The executable usually provides the basic functionality of your program, and simple programs often consist of only an executable file. You can extend the application by calling DLLs, packages, and other support files from the executable. The IDE offers two application UI models: • Single document interface (see page 1901) (SDI) • Multiple document interface (see page 1901) (MDI) In addition to the implementation model... more (see page 1902)

3.2.4.1.5.1 Setting IDE, Project, and Compiler Options

In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE. To specify various options for your project, choose **Project ▸ Options**.

Setting default project options

To change the default options that apply to all future projects, set the options in the Project Options dialog box and check the **Default** box at the bottom left of each page. All new projects will use the current options selected by default.

See Also

Project Options

3.2.4.1.5.2 Console Applications

Console applications are 32-bit programs that run without a graphical interface, in a console window. These applications typically don't require much user input and perform a limited set of functions. Any application that contains:

```
{$APPTYPE CONSOLE}
```

in the code opens a console window of its own.

To create a new console application, choose **File ▸ New ▸ Other**. Select Delphi Projects and double-click Console Application from the New Items dialog box.

The IDE then creates a project file for this type of source file and displays the Code editor.

Console applications should make sure that no exceptions escape from the program scope. Otherwise, when the program terminates, the Windows operating system displays a modal dialog with exception information. For example, your application should include exception handling such as shown in the following code:

```
program ConsoleExceptionHandling;
{$APPTYPE CONSOLE}
uses
SysUtils;
procedure ExecuteProgram;
begin
  //Program does something
  raise Exception.Create('Unforeseen exception');
end;
begin
  try
    ExecuteProgram;
  except
//Handle error condition
    WriteIn('Program terminated due to an exception');
    //Set ExitCode <> 0 to flag error condition (by convention)
```

```
    ExitCode := 1;
  end;
end.
```

Users can terminate console applications in one of the following ways:

- Click the Close (X) button.
- Press `Ctrl+C`.
- Press `Ctrl+Break`.
- Log off.

Depending on which way the user chooses, the application is terminated forcefully, the process is not shut down cleanly, and the finalization section isn't run. Use the Windows API *SetConsoleCtrlHandler* function for options for handling these user termination requests.

See Also

TApplication

3.2.4.1.5.3 MDI Applications

To create a new MDI application using a wizard:

1. Choose **File** ▸ **New** ▸ **Other** to bring up the New Items dialog.
2. Click on the Projects page and double-click MDI Application.
3. Click OK.

MDI applications require more planning and are somewhat more complex to design than SDI applications. MDI applications spawn child windows that reside within the client window; the main form contains child forms. Set the *FormStyle* property of the *TForm* object to specify whether a form is a child (*fsMDIChild*) or main form (*fsMDIForm*). It is a good idea to define a base class for your child forms and derive each child form from this class, to avoid having to reset the child form's properties.

MDI applications often include a Window pop-up on the main menu that has items such as Cascade and Tile for viewing multiple windows in various styles. When a child window is minimized, its icon is located in the MDI parent form.

To create a new MDI application without using a wizard:

1. Create the main window form or MDI parent window. Set its *FormStyle* property to *fsMDIForm*.
2. Create a menu for the main window that includes **File** ▸ **Open**, **File** ▸ **Save**, and Window which has Cascade, Tile, and Arrange All items.
3. Create the MDI child forms and set their *FormStyle* properties to *fsMDIChild*.

See Also

TForm

TScreen

TApplication

3.2.4.1.5.4 SDI Applications

To create a new SDI application:

1. Choose **File** ▸ **New** ▸ **Other** to bring up the New Items dialog.
2. Click on the Projects page and double-click SDI Application.
3. Click OK.

By default, the *FormStyle* property of your *Form* object is set to fsNormal, so that the IDE assumes that all new applications are SDI applications.

See Also

TForm

TScreen

TApplication

3.2.4.1.5.5 User Interface Models

Any form can be implemented as a single document interface (SDI) or multiple document interface (MDI) form. An SDI application (see page 1901) normally contains a single document view. In an MDI application, more than one document or child window can be opened within a single parent window. This is common in applications such as spreadsheets or word processors.

For more information on developing the UI for an application, see Developing the application user interface (see page 1983).

See Also

TForm

TScreen

TApplication

3.2.4.1.5.6 GUI Applications

A graphical user interface (GUI) application is one that is designed using graphical features such as windows, menus, dialog boxes, and features that make the application easy to use. When you compile a GUI application, an executable file with start-up code is created. The executable usually provides the basic functionality of your program, and simple programs often consist of only an executable file. You can extend the application by calling DLLs, packages, and other support files from the executable.

The IDE offers two application UI models:

- Single document interface (see page 1901) (SDI)
- Multiple document interface (see page 1901) (MDI)

In addition to the implementation model of your applications, the design-time behavior of your project and the runtime behavior of your application can be manipulated by setting project options in the IDE.

See Also

TForm

TScreen

TApplication

3.2.4.1.6 Service applications

Topics

Name	Description
Debugging Service Applications (see page 1903)	You can debug service applications by attaching to the service application process when it is already running (that is, by starting the service first, and then attaching to the debugger). To attach to the service application process, choose **Run** ▶ **Attach To Process**, and select the service application in the resulting dialog. In some cases, this approach may fail, due to insufficient rights. If that happens, you can use the Service Control Manager to enable your service to work with the debugger:

Service Applications (see page 1903)	Service applications take requests from client applications, process those requests, and return information to the client applications. They typically run in the background, without much user input. A Web, FTP, or e-mail server is an example of a service application.
Service Name Properties (see page 1906)	The VCL provides classes for creating service applications on the Windows platform (not available for cross-platform applications). These include *TService* and *TDependency*. When using these classes, the various name properties can be confusing. This topic describes the differences. Services have user names (called Service start names) that are associated with passwords, display names for display in manager and editor windows, and actual names (the name of the service). Dependencies can be services or they can be load ordering groups. They also have names and display names. And because service objects are derived from *TComponent*, they inherit the *Name... more (see page 1906)*
Service Threads (see page 1907)	Each service has its own thread (TServiceThread), so if your service application implements more than one service you must ensure that the implementation of your services is thread-safe. *TServiceThread* is designed so that you can implement the service in the *TServiceOnExecute* event handler. The service thread has its own Execute method which contains a loop that calls the service's *OnStart* and *OnExecute* handlers before processing new requests. Because service requests can take a long time to process and the service application can receive simultaneous requests from more than one client, it is more efficient to spawn a new thread... more (see page 1907)

3.2.4.1.6.1 Debugging Service Applications

You can debug service applications by attaching to the service application process when it is already running (that is, by starting the service first, and then attaching to the debugger). To attach to the service application process, choose **Run ▸ Attach To Process**, and select the service application in the resulting dialog.

In some cases, this approach may fail, due to insufficient rights. If that happens, you can use the Service Control Manager to enable your service to work with the debugger:

To debug:

1. First create a key called **Image File Execution Options** in the following registry location:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion
```

2. Create a subkey with the same name as your service (for example, MYSERV.EXE). To this subkey, add a value of type REG_SZ, named Debugger. Use the full path to bds.exe as the string value.
3. In the Services control panel applet, select your service, click Startup and check Allow Service to Interact with Desktop.

On Windows NT systems, you can use another approach for debugging service applications. However, this approach can be tricky, because it requires short time intervals:

For Windows NT:

1. First, launch the application in the debugger. Wait a few seconds until it has finished loading.
2. Quickly start the service from the Control Panel or from the command line:

```
start MyServ
```

You must launch the service quickly (within 15-30 seconds of application startup) because the application will terminate if no service is launched.

3.2.4.1.6.2 Service Applications

Service applications take requests from client applications, process those requests, and return information to the client applications. They typically run in the background, without much user input. A Web, FTP, or e-mail server is an example of a service application.

To create an application that implements a Win32 service:

1. Choose **File** ▸ **New** ▸ **Other**, and double-click Service Application in the New Items dialog box. This adds a global variable named *Application* to your project, which is of type *TServiceApplication*.
2. A Service window appears that corresponds to a service (*TService*). Implement the service by setting its properties and event handlers in the **Object Inspector**.
3. You can add additional services to your service application by choosing **File** ▸ **New** ▸ **Other**, and double-click Service in the New Items dialog box. Do not add services to an application that is not a service application. While a *TService* object can be added, the application will not generate the requisite events or make the appropriate Windows calls on behalf of the service.
4. Once your service application is built, you can install its services with the Service Control Manager (SCM). Other applications can then launch your services by sending requests to the SCM.

To install your application's services, run it using the /INSTALL option. The application installs its services and exits, giving a confirmation message if the services are successfully installed. You can suppress the confirmation message by running the service application using the /SILENT option.

To uninstall the services, run it from the command line using the /UNINSTALL option. (You can also use the /SILENT option to suppress the confirmation message when uninstalling).

Note: This service has a *TServerSocket* whose port is set to 80. This is the default port for Web browsers to make requests to Web servers and for Web servers to make responses to Web browsers. This particular example produces a text document in the C:\Temp directory called WebLog*xxx*.log (where *xxx* is the ThreadID). There should be only one server listening on any given port, so if you have a Web server, you should make sure that it is not listening (the service is stopped).

To see the results: open up a Web browser on the local machine and for the address, type 'localhost' (with no quotes). The browser will time out eventually, but you should now have a file called Weblog*xxx*.log in the C:\Temp directory.

To create the example:

1. Choose **File** ▸ **New** ▸ **Other** and select Service Application from the New Items dialog box. The Service1 window appears.
2. From the Internet category of the **Tool palette**, add a ServerSocket component to the service window (Service1).
3. Add a private data member of type TMemoryStream to the TService1 class. The interface section of your unit should now look like this:

```
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, SvcMgr, Dialogs,
  ScktComp;
type
    TService1 = class(TService)
    ServerSocket1: TServerSocket;
    procedure ServerSocket1ClientRead(Sender: TObject;
      Socket: TCustomWinSocket);
    procedure Service1Execute(Sender: TService);
  private
    { Private declarations }
    Stream: TMemoryStream; // Add this line here
  public
    function GetServiceController: PServiceController; override;
    { Public declarations }
  end;
var
  Service1: TService1;
//---------------------------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <SysUtils.hpp>
#include <Classes.hpp>
#include <SvcMgr.hpp>
#include <ScktComp.hpp>
//---------------------------------------------------------------------------
```

```
class TService1 : public TService
{
__published:
TServerSocket *ServerSocket1;
private:
TMemoryStream *Stream; // add this line here
public:
__fastcall TService1(TComponent* Owner);
PServiceController __fastcall GetServiceController(void);
friend void __stdcall ServiceController(unsigned CtrlCode);
};
//---------------------------------------------------------------------------
extern PACKAGE TService1 *Service1;
//---------------------------------------------------------------------------
#endif
```

4. Select ServerSocket1, the component you added in step 1. In the **Object Inspector**, double-click the OnClientRead event and add the following event handler:

```
procedure TService1.ServerSocket1ClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
  Buffer: PChar;
begin
  Buffer := nil;
while Socket.ReceiveLength > 0 do begin
    Buffer := AllocMem(Socket.ReceiveLength);
    try
      Socket.ReceiveBuf(Buffer^, Socket.ReceiveLength);
      Stream.Write(Buffer^, StrLen(Buffer));
    finally
                  FreeMem(Buffer);
    end;
  Stream.Seek(0, soFromBeginning);
  Stream.SaveToFile('c:\Temp\Weblog' + IntToStr(ServiceThread.ThreadID) + '.log');
  end;
end;
void __fastcall TService1::ServerSocket1ClientRead(TObject *Sender,
TCustomWinSocket *Socket)
{
char *Buffer = NULL;
int len = Socket->ReceiveLength();
while (len > 0)
{
try
{
Buffer = (char *)malloc(len);
Socket->ReceiveBuf((void *)Buffer, len);
Stream->Write(Buffer, len);
}
__finally
{
free(Buffer);
}
Stream->Seek(0, soFromBeginning);
AnsiString LogFile = "C:\\Temp\\WebLog";
LogFile = LogFile + IntToStr(ServiceThread->ThreadID) + ".log";
Stream->SaveToFile(LogFile);
}
}
```

5. Finally, select Service1 by clicking in the window's client area (but not on the ServiceSocket). In the **Object Inspector**, double click the OnExecute event and add the following event handler:

```
procedure TService1.Service1Execute(Sender: TService);
begin
  Stream := TMemoryStream.Create;
```

```
    try
      ServerSocket1.Port := 80; // WWW port
      ServerSocket1.Active := True;
      while not Terminated do begin
        ServiceThread.ProcessRequests(True);
      end;
      ServerSocket1.Active := False;
    finally
      Stream.Free;
    end;
  end;
  void __fastcall TService1::Service1Execute(TService *Sender)
  {
  Stream = new TMemoryStream();
  try
  {
  ServerSocket1->Port = 80; // WWW port
  ServerSocket1->Active = true;
  while (!Terminated)
  ServiceThread->ProcessRequests(true);
  ServerSocket1->Active = false;
  }
  __finally
  {
  delete Stream;
  }
  }
```

When writing your service application, you should be aware of:

- Service threads (see page 1907)
- Service name properties (see page 1906)
- Debugging service applications (see page 1903)

Note: Service applications are not available for cross-platform applications.

See Also

TServiceApplication

TService

3.2.4.1.6.3 Service Name Properties

The VCL provides classes for creating service applications on the Windows platform (not available for cross-platform applications). These include *TService* and *TDependency*. When using these classes, the various name properties can be confusing. This topic describes the differences.

Services have user names (called Service start names) that are associated with passwords, display names for display in manager and editor windows, and actual names (the name of the service). Dependencies can be services or they can be load ordering groups. They also have names and display names. And because service objects are derived from *TComponent*, they inherit the *Name* property. The following sections summarize the name properties.

TDependency properties

The *TDependency DisplayName* is both a display name and the actual name of the service. It is nearly always the same as the *TDependency Name* property.

TService name properties

The *TService Name* property is inherited from *TComponent*. It is the name of the component, and is also the name of the service. For dependencies that are services, this property is the same as the *TDependency Name* and *DisplayName* properties.

TService's DisplayName is the name displayed in the Service Manager window. This often differs from the actual service name

(*TService.Name, TDependency.DisplayName, TDependency.Name*). Note that the *DisplayName* for the Dependency and the *DisplayName* for the Service usually differ.

Service start names are distinct from both the service display names and the actual service names. A *ServiceStartName* is the user name input on the Start dialog selected from the Service Control Manager.

See Also

TService

TServiceApplication

TDependency

3.2.4.1.6.4 Service Threads

Each service has its own thread (TServiceThread), so if your service application implements more than one service you must ensure that the implementation of your services is thread-safe. *TServiceThread* is designed so that you can implement the service in the *TServiceOnExecute* event handler. The service thread has its own Execute method which contains a loop that calls the service's *OnStart* and *OnExecute* handlers before processing new requests.

Because service requests can take a long time to process and the service application can receive simultaneous requests from more than one client, it is more efficient to spawn a new thread (derived from TThread, not *TServiceThread*) for each request and move the implementation of that service to the new thread's *Execute* method. This allows the service thread's *Execute* loop to process new requests continually without having to wait for the service's *OnExecute* handler to finish. The following example demonstrates.

Note: This service beeps every 500 milliseconds from within the standard thread. It handles pausing, continuing, and stopping of the thread when the service is told to pause, continue, or stop.

To create the example:

1. Choose **File** ▶ **New** ▶ **Other** and double-click Service Application in the New Items dialog. The Service1 window appears.
2. In the interface section of your unit, declare a new descendant of TThread named TSparkyThread. This is the thread that does the work for your service. The declaration should appear as follows:

```
TSparkyThread = class(TThread)
  public
    procedure Execute; override;
end;
```

```
class TSparkyThread : public TThread
{
private:
protected:
  void __fastcall Execute();
public:
__fastcall TSparkyThread(bool CreateSuspended);
};
```

3. In the implementation section of your unit, create a global variable for a TSparkyThread instance:

```
var
  SparkyThread: TSparkyThread;
```

```
TSparkyThread *SparkyThread;// Add this code as the constructor
__fastcall TSparkyThread::TSparkyThread(bool CreateSuspended)
: TThread(CreateSuspended)
{
}
```

4. In the implementation section for the TSparkyThread Execute method (the thread function), add the following code:

```
procedure TSparkyThread.Execute;
begin
  while not Terminated do
```

```
  begin
    Beep;
    Sleep(500);
  end;
end;
void __fastcall TSparkyThread::Execute()
{
  while (!Terminated)
  {
    Beep();
    Sleep(500);
  }
}
```

5. Select the Service window (Service1), and double-click the OnStart event in the **Object Inspector**. Add the following OnStart event handler:

```
procedure TService1.Service1Start(Sender: TService; var Started: Boolean);
begin
  SparkyThread := TSparkyThread.Create(False);
  Started := True;
end;
void __fastcall TService1::Service1Start(TService *Sender, bool &Started)
{
  SparkyThread = new TSparkyThread(false);
  Started = true;
}
```

6. Double-click the OnContinue event in the **Object Inspector**. Add the following OnContinue event handler:

```
procedure TService1.Service1Continue(Sender: TService; var Continued: Boolean);
begin
  SparkyThread.Resume;
  Continued := True;
end;
void __fastcall TService1::Service1Continue(TService *Sender, bool &Continued)
{
  SparkyThread->Resume();
  Continued = true;
}
```

7. Double-click the OnPause event in the **Object Inspector**. Add the following OnPause event handler:

```
procedure TService1.Service1Pause(Sender: TService; var Paused: Boolean);
begin
  SparkyThread.Suspend;
  Paused := True;
end;
void __fastcall TService1::Service1Pause(TService *Sender, bool &Paused)
{
  SparkyThread->Suspend();
  Paused = true;
}
```

8. Finally, double-click the OnStop event in the **Object Inspector** and add the following OnStop event handler:

```
procedure TService1.Service1Stop(Sender: TService; var Stopped: Boolean);
begin
  SparkyThread.Terminate;
  Stopped := True;
end;
void __fastcall TService1::Service1Stop(TService *Sender, bool &Stopped)
{
  SparkyThread->Terminate();
  Stopped = true;
}
```

When developing server applications, choosing to spawn a new thread depends on the nature of the service being provided, the anticipated number of connections, and the expected number of processors on the computer running the service.

See Also

TServiceThread

TService

3.2.4.1.7 Using data modules

Topics

Name	Description
Accessing a Data Module from a Form (see page 1910)	To associate visual controls on a form with a data module, you must first add the data module to the form's **uses** clause. You can do this in several ways: • In the Code editor, open the form's unit file and add the name of the data module to the **uses** clause in the **interface** section. • Click the form's unit file, choose **File ▶ Use Unit**, and enter the name of the module or pick it from the list box in the Use Unit dialog. • For database components, in the data module click a dataset or query component to open the... more (see page 1910)
Adding a Remote Data Module to an Application Server Project (see page 1910)	Some editions of Delphi allow you to add *remote data modules* to application server projects. A remote data module has an interface that clients in a multi-tiered application can access across networks.
Creating and Editing Standard Data Modules (see page 1911)	To create a standard data module for a project, choose **File ▶ New ▶ Other ▶ Delphi Projects ▶ Delphi Files ▶ Data Module**. The IDE opens a data module container on the desktop, displays the unit file for the new module in the Code editor, and adds the module to the current project. At design time, a data module looks like a standard form with a white background and no alignment grid. As with forms, you can place nonvisual components from the **Tool palette** onto a module, and edit their properties in the **Object Inspector**. You can resize a data module to... more (see page 1911)
Creating Business Rules in a Data Module (see page 1911)	Besides writing event handlers for the components in a data module, you can code methods directly in the unit file for a data module. These methods can be applied to the forms that use the data module as business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping. You might call the procedure from an event handler for a component in the data module. The prototypes for the procedures and functions you write for a data module should appear in the module's **type** declaration:
Using Data Modules (see page 1912)	A data module is like a special form that contains nonvisual components. All the components in a data module *could* be placed on ordinary forms alongside visual controls. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then data modules provide a convenient organizational tool. There are several types of data modules, including standard, remote, Web modules, applet modules, and services, depending on which edition of Delphi you have. Each type of data module serves a special purpose. • Standard... more (see page 1912)
Naming a Data Module and Its Unit File (see page 1912)	The title bar of a data module displays the module's name. The default name for a data module is "DataModule*N*" where *N* is a number representing the lowest unused unit number in a project. For example, if you start a new project, and add a module to it before doing any other application building, the name of the module defaults to "DataModule2." The corresponding unit file for *DataModule2* defaults to "Unit2." You should rename your data modules and their corresponding unit files at design time to make them more descriptive. You should especially rename data modules you add... more (see page 1912)

Placing and Naming Components (see page 1913)	You place nonvisual components in a data module just as you place visual components on a form. Click the desired component on the appropriate category of the **Tool palette**, then click in the data module to place the component. You cannot place visual controls, such as grids, on a data module. If you attempt it, you receive an error message. For ease of use, components are displayed with their names in a data module. When you first place a component, the module assigns it a generic name that identifies what kind of component it is, followed by a 1.... more (see page 1913)
Using Component Properties and Events in a Data Module (see page 1913)	Placing components in a data module centralizes their behavior for your entire application. For example, you can use the properties of dataset components, such as *TClientDataSet*, to control the data available to the data source components that use those datasets. Setting the *ReadOnly* property to *True* for a dataset prevents users from editing the data they see in a data-aware visual control on a form. You can also invoke the Fields editor for a dataset, by double-clicking on *ClientDataSet1*, to restrict the fields within a table or query that are available to a data source and therefore to... more (see page 1913)

3.2.4.1.7.1 Accessing a Data Module from a Form

To associate visual controls on a form with a data module, you must first add the data module to the form's **uses** clause. You can do this in several ways:

- In the Code editor, open the form's unit file and add the name of the data module to the **uses** clause in the **interface** section.
- Click the form's unit file, choose **File ▸ Use Unit**, and enter the name of the module or pick it from the list box in the Use Unit dialog.
- For database components, in the data module click a dataset or query component to open the Fields editor and drag any existing fields from the editor onto the form. The IDE prompts you to confirm that you want to add the module to the form's **uses** clause, then creates controls (such as edit boxes) for the fields.

For example, if you've added the *TClientDataSet* component to your data module, double-click it to open the Fields editor. Select a field and drag it to the form. An edit box component appears.

Because the data source is not yet defined, Delphi adds a new data source component, *DataSource1*, to the form and sets the edit box's *DataSource* property to *DataSource1*. The data source automatically sets its *DataSet* property to the dataset component, *ClientDataSet1*, in the data module.

You can define the data source *before* you drag a field to the form by adding a TDataSource component to the data module. Set the data source's *DataSet* property to *ClientDataSet1*. After you drag a field to the form, the edit box appears with its *TDataSource* property already set to *DataSource1*. This method keeps your data access model cleaner.

See Also

Using Data Modules and Remote Data Modules (see page 1912)

Creating and Editing Data Modules (see page 1911)

3.2.4.1.7.2 Adding a Remote Data Module to an Application Server Project

Some editions of Delphi allow you to add *remote data modules* to application server projects. A remote data module has an interface that clients in a multi-tiered application can access across networks.

To add a remote data module to a project:

1. Choose **File ▸ New ▸ Other**.
2. Select the ActiveX page in the New Items dialog box.
3. Double-click the Remote Data Module icon to open the Remote Data Module wizard.

Once you add a remote data module to a project, use it just like a standard data module.

For more information about multi-tiered database applications, see Creating multi-tiered applications (see page 1518).

See Also

Using Data Modules and Remote Data Modules (see page 1912)

3.2.4.1.7.3 Creating and Editing Standard Data Modules

To create a standard data module for a project, choose **File** ▶ **New** ▶ **Other** ▶ **Delphi Projects** ▶ **Delphi Files** ▶ **Data Module**. The IDE opens a data module container on the desktop, displays the unit file for the new module in the Code editor, and adds the module to the current project.

At design time, a data module looks like a standard form with a white background and no alignment grid. As with forms, you can place nonvisual components from the **Tool palette** onto a module, and edit their properties in the **Object Inspector**. You can resize a data module to accommodate the components you add to it.

You can also right-click a module to display a context menu for it. The following table summarizes the context menu options for a data module.

Context menu options for data modules

Menu item	Purpose
Edit	Displays a context menu with which you can cut, copy, paste, delete, and select the components in the data module.
Position	Aligns nonvisual components to the module's invisible grid (*Align To Grid*) *or* according to criteria you supply in the Alignment dialog box (*Align*).
Tab Order	Enables you to change the order that the focus jumps from component to component when you press the tab key.
Creation Order	Enables you to change the order that data access components are created at start-up.
Revert to Inherited	Discards changes made to a module inherited from another module in the Object Repository, and reverts to the originally inherited module.
Add to Repository	Stores a link to the data module in the Object Repository.
View as Text	Displays the text representation of the data module's properties.
Text DFM	Toggles between the formats (binary or text) in which this particular form file is saved.

See Also

Using Data Modules and Remote Data Modules (see page 1912)

3.2.4.1.7.4 Creating Business Rules in a Data Module

Besides writing event handlers for the components in a data module, you can code methods directly in the unit file for a data module. These methods can be applied to the forms that use the data module as business rules. For example, you might write a procedure to perform month-, quarter-, or year-end bookkeeping. You might call the procedure from an event handler for a component in the data module.

The prototypes for the procedures and functions you write for a data module should appear in the module's **type** declaration:

```
type
  TCustomerData = class(TDataModule)
    Customers: TClientDataSet;
    Orders: TClientDataSet;
    .
    .
    .
  private
    { Private declarations }
  public
    { Public declarations }
```

```
    procedure LineItemsCalcFields(DataSet: TDataSet); { A procedure you add }
  end;
var
  CustomerData: TCustomerData;
```

The procedures and functions you write should follow in the implementation section of the code for the module.

See Also

Designing Database Applications (see page 1566)

3.2.4.1.7.5 Using Data Modules

A data module is like a special form that contains nonvisual components. All the components in a data module *could* be placed on ordinary forms alongside visual controls. But if you plan on reusing groups of database and system objects, or if you want to isolate the parts of your application that handle database connectivity and business rules, then data modules provide a convenient organizational tool.

There are several types of data modules, including standard, remote, Web modules, applet modules, and services, depending on which edition of Delphi you have. Each type of data module serves a special purpose.

- Standard data modules are particularly useful for single- and two-tiered database applications, but can be used to organize the nonvisual components in any application. For more information, see Creating and Editing Data Modules (see page 1911).
- Remote data modules form the basis of an application server in a multi-tiered database application. They are not available in all editions. In addition to holding the nonvisual components in the application server, remote data modules expose the interface that clients use to communicate with the application server. For more information about using them, see Adding a remote data module to an application server project (see page 1910).
- Web modules form the basis of Web server applications. In addition to holding the components that create the content of HTTP response messages, they handle the dispatching of HTTP messages from client applications. See Creating Internet Server Applications (see page 2251) for more information about using Web modules.
- Applet modules form the basis of control panel applets. In addition to holding the nonvisual controls that implement the control panel applet, they define the properties that determine how the applet's icon appears in the control panel and include the events that are called when users execute the applet.
- Services encapsulate individual services in an NT service application. In addition to holding any nonvisual controls used to implement a service, services include the events that are called when the service is started or stopped. For more information about services, see Service Applications (see page 1903).

See Also

Designing Database Applications (see page 1566)

Creating and Editing Data Modules (see page 1911)

3.2.4.1.7.6 Naming a Data Module and Its Unit File

The title bar of a data module displays the module's name. The default name for a data module is "DataModule*N*" where *N* is a number representing the lowest unused unit number in a project. For example, if you start a new project, and add a module to it before doing any other application building, the name of the module defaults to "DataModule2." The corresponding unit file for *DataModule2* defaults to "Unit2."

You should rename your data modules and their corresponding unit files at design time to make them more descriptive. You should especially rename data modules you add to the Object Repository to avoid name conflicts with other data modules in the Repository or in applications that use your modules.

To rename a data module:

1. Select the module.

2. Edit the Name property for the module in the **Object Inspector**.

The new name for the module appears in the title bar when the *Name* property in the **Object Inspector** no longer has focus.

Changing the name of a data module at design time changes its variable name in the interface section of code. It also changes any use of the type name in procedure declarations. You must manually change any references to the data module in code you write.

To rename a unit file for a data module, select the unit file.

See Also

Designing Database Applications (see page 1566)

Creating and Editing Data Modules (see page 1911)

3.2.4.1.7.7 Placing and Naming Components

You place nonvisual components in a data module just as you place visual components on a form. Click the desired component on the appropriate category of the **Tool palette**, then click in the data module to place the component. You cannot place visual controls, such as grids, on a data module. If you attempt it, you receive an error message.

For ease of use, components are displayed with their names in a data module. When you first place a component, the module assigns it a generic name that identifies what kind of component it is, followed by a 1. For example, the *TDataSource* component adopts the name *DataSource1*. This makes it easy to select specific components whose properties and methods you want to work with.

You may still want to name a component a different name that reflects the type of component and what it is used for.

To change the name of a component in a data module:

1. Select the component.
2. Edit the component's *Name* property in the **Object Inspector**.

The new name for the component appears under its icon in the data module as soon as the *Name* property in the **Object Inspector** no longer has focus.

For example, suppose your database application uses the CUSTOMER table. To access the table, you need a minimum of two data access components: a data source component (*TDataSource*) and a table component (*TClientDataSet*). When you place these components in your data module, the module assigns them the names *DataSource1* and *ClientDataSet1*. To reflect the type of component and the database they access, CUSTOMER, you could change these names to *CustomerSource* and *CustomerTable*.

See Also

Designing Database Applications (see page 1566)

Creating and Editing Data Modules (see page 1911)

3.2.4.1.7.8 Using Component Properties and Events in a Data Module

Placing components in a data module centralizes their behavior for your entire application. For example, you can use the properties of dataset components, such as *TClientDataSet*, to control the data available to the data source components that use those datasets. Setting the *ReadOnly* property to *True* for a dataset prevents users from editing the data they see in a data-aware visual control on a form. You can also invoke the Fields editor for a dataset, by double-clicking on *ClientDataSet1*, to restrict the fields within a table or query that are available to a data source and therefore to the data-aware controls on forms. The properties you set for components in a data module apply consistently to all forms in your application that use the module.

In addition to properties, you can write event handlers for components. For example, a TDataSource component has three possible events: *OnDataChange*, *OnStateChange*, and *OnUpdateData*. A *TClientDataSet* component has over 20 potential events. You can use these events to create a consistent set of business rules that govern data manipulation throughout your

application.

See Also

Designing Database Applications (see page 1566)

Creating Business Rules in a Data Module (see page 1911)

3.2.4.1.8 Using the Object Repository

Topics

Name	Description
Adding Items to the Object Repository (see page 1915)	You can add your own projects, forms, frames, and data modules to those already available in the Object Repository.
Copying an Item (see page 1915)	Choose Copy to make an exact copy of the selected item and add the copy to your project. Future changes made to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item. Copy is the only option available for project templates.
Inheriting an Item (see page 1915)	Choose Inherit to derive a new class from the selected item and add the new class to your project. When you recompile your project, any changes that have been made to the item Will be reflected in your derived class, in addition to changes you make to the item in your project. Changes made to your derived class do not affect the shared item in the Object Repository. Inherit is available for forms, dialog boxes, and data modules, but not for project templates. It is the *only* option available for reusing items within the same project.
Modifying Shared Items (see page 1916)	If you modify an item in the Object Repository, your changes will affect all future projects that use the item as well as existing projects that have added the item with the Use or Inherit option. To avoid propagating changes to other projects, you have several alternatives: • Copy the item and modify it in your current project only. • Copy the item to the current project, modify it, then add it to the Repository under a different name. • Create a component, DLL, component template, or frame from the item. If you create a component or DLL, you can share it with... more (see page 1916)
Sharing Items Within a Project (see page 1916)	You can share items *within* a project without adding them to the Object Repository. When you open the New Items dialog box (**File** ▶ **New** ▶ **Other**), you"ll see a page tab with the name of the current project. This page lists all the forms, dialog boxes, and data modules in the project. You can derive a new item from an existing item and customize it as needed.
Sharing Objects in a Team Environment (see page 1916)	You can share objects with your workgroup or development team by making a repository available over a network.
Using an Item (see page 1916)	Choose Use when you want the selected item itself to become part of your project. Changes made to the item in your project will appear in all other projects that have added the item with the Inherit or Use option. Select this option with caution. The Use option is available for forms, dialog boxes, and data modules.
Using an Object Repository Item in a Project (see page 1917)	To access items in the Object Repository, choose **File** ▶ **New** ▶ **Other**. The New Items dialog appears, showing all the items available. Depending on the type of item you want to use, you have up to three options for adding the item to your project: • Copy (see page 1915) • Inherit (see page 1915) • Use (see page 1916)
Using Project Templates (see page 1917)	Templates are predesigned projects that you can use as starting points for your own work.

Using the Object Repository (see page 1917)	The Object Repository (**Tools** ▶ **Options** ▶ **Repository (under Translation Tools Options)**) makes it easy share forms, dialog boxes, frames, and data modules. It also provides templates for new projects and wizards that guide the user through the creation of forms and projects. The Object Repository is maintained in DELPHI32.DRO (by default in the BIN directory), a text file that contains references to the items that appear in the Repository and New Items dialogs.

3.2.4.1.8.1 Adding Items to the Object Repository

You can add your own projects, forms, frames, and data modules to those already available in the Object Repository.

To add an item to the Object Repository

1. If the item is a project or is in a project, open the project.
2. For a project, choose **Project** ▶ **Add To Repository**. For a form or data module, right-click the item and choose Add To Repository.
3. Type a description, title, and author.
4. Decide which page you want the item to appear on in the New Items dialog box, then type the name of the page or select it from the Page combo box. If you type the name of a page that doesn't exist, the Object Repository creates a new page.
5. Choose Browse to select an icon to represent the object in the Object Repository.
6. Choose OK.

See Also

Using the Object Repository (see page 1917)

3.2.4.1.8.2 Copying an Item

Choose Copy to make an exact copy of the selected item and add the copy to your project. Future changes made to the item in the Object Repository will not be reflected in your copy, and alterations made to your copy will not affect the original Object Repository item.

Copy is the only option available for project templates.

See Also

Using an Object Repository Item in a Project (see page 1917)

Inheriting an Item (see page 1915)

Using an Item (see page 1916)

3.2.4.1.8.3 Inheriting an Item

Choose Inherit to derive a new class from the selected item and add the new class to your project. When you recompile your project, any changes that have been made to the item Will be reflected in your derived class, in addition to changes you make to the item in your project. Changes made to your derived class do not affect the shared item in the Object Repository.

Inherit is available for forms, dialog boxes, and data modules, but not for project templates. It is the *only* option available for reusing items within the same project.

See Also

Using an Object Repository Item in a Project (see page 1917)

Using an Item (see page 1916)

Copying an Item (see page 1915)

3.2.4.1.8.4 Modifying Shared Items

If you modify an item in the Object Repository, your changes will affect all future projects that use the item as well as existing projects that have added the item with the Use or Inherit option. To avoid propagating changes to other projects, you have several alternatives:

- Copy the item and modify it in your current project only.
- Copy the item to the current project, modify it, then add it to the Repository under a different name.
- Create a component, DLL, component template, or frame from the item. If you create a component or DLL, you can share it with other developers.

See Also

Using the Object Repository (see page 1917)

Copying an Item (see page 1915)

Adding Items to the Object Repository (see page 1915)

3.2.4.1.8.5 Sharing Items Within a Project

You can share items *within* a project without adding them to the Object Repository. When you open the New Items dialog box (**File** ▸ **New** ▸ **Other**), you"ll see a page tab with the name of the current project. This page lists all the forms, dialog boxes, and data modules in the project. You can derive a new item from an existing item and customize it as needed.

See Also

Using the Object Repository (see page 1917)

3.2.4.1.8.6 Sharing Objects in a Team Environment

You can share objects with your workgroup or development team by making a repository available over a network.

To use a shared repository, all team members must select the same Shared Repository directory in the Environment Options dialog:

1. Choose **Tools** ▸ **Options** ▸ **Environment Options**.
2. On the Preferences page, locate the Shared Repository panel. In the Directory edit box, enter the directory where you want to locate the shared repository. Be sure to specify a directory that's accessible to all team members.

The first time an item is added to the Repository, a DELPHI32.DRO file is created in the Shared Repository directory if one doesn't exist already.

See Also

Using the Object Repository (see page 1917)

3.2.4.1.8.7 Using an Item

Choose Use when you want the selected item itself to become part of your project. Changes made to the item in your project will appear in all other projects that have added the item with the Inherit or Use option. Select this option with caution.

The Use option is available for forms, dialog boxes, and data modules.

See Also

Using an Object Repository Item in a Project (see page 1917)

Copying an Item (see page 1915)

Inheriting an Item (see page 1915)

3.2.4.1.8.8 Using an Object Repository Item in a Project

To access items in the Object Repository, choose **File ▸ New ▸ Other**. The New Items dialog appears, showing all the items available. Depending on the type of item you want to use, you have up to three options for adding the item to your project:

- Copy (see page 1915)
- Inherit (see page 1915)
- Use (see page 1916)

See Also

Using the Object Repository (see page 1917)

3.2.4.1.8.9 Using Project Templates

Templates are predesigned projects that you can use as starting points for your own work.

To create a new project from a template:

1. Choose **File ▸ New ▸ Other** to display the New Items dialog box.
2. Choose the Projects tab.
3. Select the project template you want and choose OK.
4. In the Select Directory dialog, specify a directory for the new project's files.

The template files are copied to the specified directory, where you can modify them. The original project template is unaffected by your changes.

To add projects and project templates to the Object Repository, see Adding items to the Object Repository (see page 1915).

See Also

Using the Object Repository (see page 1917)

3.2.4.1.8.10 Using the Object Repository

The Object Repository (**Tools ▸ Options ▸ Repository (under Translation Tools Options)**) makes it easy share forms, dialog boxes, frames, and data modules. It also provides templates for new projects and wizards that guide the user through the creation of forms and projects. The Object Repository is maintained in DELPHI32.DRO (by default in the BIN directory), a text file that contains references to the items that appear in the Repository and New Items dialogs.

See Also

Sharing Items Within A Project (see page 1916)

Adding Items to the Object Repository (see page 1915)

Sharing Objects in a Team Environment (see page 1916)

Using an Object Repository Item in a Project (see page 1917)

Using Project Templates (see page 1917)

Modifying a Shared Item (see page 1916)

Reusing Components and Groups of Components (see page 2000)

3.2.4.1.9 Writing applications using COM

Topics

Name	Description
Writing Applications Using COM (see page 1918)	COM is the Component Object Model, a Windows-based distributed object architecture designed to provide object interoperability using predefined routines called interfaces. COM applications use objects that are implemented by a different process or, if you use DCOM, on a separate machine. You can also use COM+, ActiveX and Active Server Pages. COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients... more (see page 1918)

3.2.4.1.9.1 Writing Applications Using COM

COM is the Component Object Model, a Windows-based distributed object architecture designed to provide object interoperability using predefined routines called interfaces. COM applications use objects that are implemented by a different process or, if you use DCOM, on a separate machine. You can also use COM+, ActiveX and Active Server Pages.

COM is a language-independent software component model that enables interaction between software components and applications running on a Windows platform. The key aspect of COM is that it enables communication between components, between applications, and between clients and servers through clearly defined interfaces. Interfaces provide a way for clients to ask a COM component which features it supports at runtime. To provide additional features for your component, you simply add an additional interface for those features.

Using COM and DCOM

Various classes and wizards that make it easy to create COM, OLE, or ActiveX applications. You can create COM clients or servers that implement COM objects, Automation servers (including Active Server Objects), ActiveX controls, or ActiveForms. COM also severs as the basis for other technologies such as Automation, ActiveX controls, Active Documents, and Active Directories.

Using Delphi to create COM-based applications offers a wide range of possibilities, from improving software design by using interfaces internally in an application, to creating objects that can interact with other COM-based API objects on the system, such as the Win9x Shell extensions and DirectX multimedia support. Applications can access the interfaces of COM components that exist on the same computer as the application or that exist on another computer on the network using a mechanism called Distributed COM (DCOM).

For more information on COM and Active X controls, see Overview of COM technologies (see page 1385)Creating an ActiveX Control and Distributing a Client Application as an ActiveX Control (see page 1530).

For more information on DCOM, see Using DCOM connections (see page 1546).

Using MTS and COM+

COM applications can be augmented with special services for managing objects in a large distributed environment. These services include transaction services, security, and resource management supplied by Microsoft Transaction Server (MTS) on versions of Windows prior to Windows 2000) or COM+ (for Windows 2000 and later).

For more information on MTS and COM+, see Creating MTS or COM+ objects and Using transactional data modules (see page 1546).

See Also

TCOMConnection

TMTSAutoObject

TMTSDataModule

3.2.4.1.10 Writing database applications

Topics

Name	Description
Distributing Database Applications (see page 1919)	You can create distributed database applications using a coordinated set of components. Distributed database applications can be built on a variety of communications protocols, including DCOM, TCP/IP, and SOAP. For more information about building distributed database applications, see Creating Multi-tiered Applications. (see page 1518) Distributing database applications often requires you to distribute the Borland Database Engine (BDE) in addition to the application files. For information on deploying the BDE, see Deploying Database Applications. (see page 1948)
Writing Database Applications (see page 1919)	You can create advanced database applications using tools to connect to SQL servers and databases such as Oracle, Sybase, InterBase, MySQL, MS-SQL, Informix, PostgreSQL, and DB2 while providing transparent data sharing between applications. The **Tool palette** includes many components for accessing databases and representing the information they contain. The database components are grouped according to the data access mechanism and function. ***Database pages on the Tool palette***

3.2.4.1.10.1 Distributing Database Applications

You can create distributed database applications using a coordinated set of components. Distributed database applications can be built on a variety of communications protocols, including DCOM, TCP/IP, and SOAP.

For more information about building distributed database applications, see Creating Multi-tiered Applications. (see page 1518)

Distributing database applications often requires you to distribute the Borland Database Engine (BDE) in addition to the application files. For information on deploying the BDE, see Deploying Database Applications. (see page 1948)

See Also

TDataSet

TDatabase

3.2.4.1.10.2 Writing Database Applications

You can create advanced database applications using tools to connect to SQL servers and databases such as Oracle, Sybase, InterBase, MySQL, MS-SQL, Informix, PostgreSQL, and DB2 while providing transparent data sharing between applications.

The **Tool palette** includes many components for accessing databases and representing the information they contain. The database components are grouped according to the data access mechanism and function.

Database pages on the Tool palette

Palette page	Contents
BDE	Components that use the Borland Database Engine (BDE), a large API for interacting with databases. The BDE supports the broadest range of functions and comes with the most supporting utilities including Database Desktop and Database Explorer . See Using the Borland Database Engine (see page 1643) for details.
ADO	Components that use ActiveX Data Objects (ADO), developed by Microsoft, to access database information. Many ADO drivers are available for connecting to different database servers. ADO-based components let you integrate your application into an ADO-based environment. See Working with ADO Components (see page 1494) for details.

dbExpress	Cross-platform components that use dbExpress to access database information. dbExpress drivers provide fast access to databases but need to be used with *TClientDataSet* and *TDataSetProvider* to perform updates. See Using Unidirectional Datasets (see page 1823) for details.
InterBase	Components that access InterBase databases directly, without going through a separate engine layer.
Data Access	Components that can be used with any data access mechanism such as *TClientDataSet* and *TDataSetProvider*. See Using Client Datasets: Overview (see page 1740) for information about client datasets. See Using Provider Components (see page 1819) for information about providers.
Data Controls	Data-aware controls that can access information from a data source. See Using Data Controls (see page 1778) for details.

When designing a database application, you must decide which data access mechanism to use. Each data access mechanism differs in its range of functional support, the ease of deployment, and the availability of drivers to support different database servers.

Refer to Designing database applications (see page 1566) for details on what type of database support is available and considerations when designing database client applications and application servers.

Note: Not all editions of Delphi include database support.

3.2.4.2 Creating international applications

Topics

Name	Description
Creating International Applications: Overview (see page 1923)	This topic discusses guidelines for writing applications that you plan to distribute to an international market. By planning ahead, you can reduce the amount of time and code necessary to make your application function in its foreign market as well as in its domestic market. The following topics are discussed in this section: • Internationalization and localization (see page 1923) • Internationalizing applications (see page 1924) • Localizing applications (see page 1934)
Internationalization and Localization (see page 1923)	To create an application that you can distribute to foreign markets, there are two major steps that need to be performed: • Internationalization (see page 1923) • Localization (see page 1924) If your edition includes the Translation Tools, you can use the them to manage localization.
Internationalization (see page 1923)	Internationalization is the process of enabling your program to work in multiple locales. A locale is the user's environment, which includes the cultural conventions of the target country as well as the language. Windows supports many locales, each of which is described by a language and country pair.
Localization (see page 1924)	Localization is the process of translating an application so that it functions in a specific locale. In addition to translating the user interface, localization may include functionality customization. For example, a financial application may be modified for the tax laws in different countries.

Internationalizing Applications (see page 1924)	You need to complete the following steps to create internationalized applications: • Enable your code (see page 1924) to handle strings from international character sets. • Design your user interface (see page 1929) to accommodate the changes that result from localization. • Isolate all resources (see page 1931) that need to be localized.
Enabling Application Code (see page 1924)	You must make sure that the code in your application can handle the strings it will encounter in the various target locales. To do this, you must consider the following: • Character sets (see page 1924) • OEM and ANSI character sets (see page 1925) • Multibyte character sets (see page 1925) • Wide characters • Locale-specific features (see page 1928)
Character Sets (see page 1924)	The Western editions (including English, French, and German) of Windows use the ANSI Latin-1 (1252) character set. However, other editions of Windows use different character sets. For example, the Japanese version of Windows uses the Shift-JIS character set (code page 932), which represents Japanese characters as multibyte character codes. There are generally three types of characters sets: • Single-byte • Multibyte • Wide characters Windows and Linux both support single-byte and multibyte character sets as well as Unicode. With a single-byte character set, each byte in a string represents one character. The ANSI character set used by many western operating systems is a... more (see page 1924)
OEM and ANSI Character Sets (see page 1925)	It is sometimes necessary to convert between the Windows character set (ANSI) and the character set specified by the code page of the user's machine (called the OEM character set).
Multibyte Character Sets (MBCS) (see page 1925)	The ideographic character sets used in Asia cannot use the simple 1:1 mapping between characters in the language and the one byte (8-bit) **AnsiChar** type. These languages have too many characters to be represented using the single-byte **AnsiChar**. Instead, a multibyte character set string can contain one or more bytes per character. A multibyte character set provides a way to encode characters outside the standard ANSI range into single byte strings of **AnsiChar**. The lead byte of every multibyte character code is taken from a reserved range that depends on the specific character set. The second and subsequent... more (see page 1925)
Including Bi-directional Functionality in Applications (see page 1926)	Some languages do not follow the left to right reading order commonly found in western languages, but rather read words right to left and numbers left to right. These languages are termed bi-directional (BiDi) because of this separation. The most common bi-directional languages are Arabic and Hebrew, although other Middle East languages are also bi-directional. *TApplication* has two properties, *BiDiKeyboard* and *NonBiDiKeyboard*, that allow you to specify the keyboard layout. In addition, the VCL supports bi-directional localization through the *BiDiMode* and *ParentBiDiMode* properties. **Note:** Bi-directional properties are not available for cross-platform applications. The BiDiMode property controls the reading order... more (see page 1926)
ParentBiDiMode Property (see page 1927)	*ParentBiDiMode* is a Boolean property. When *True* (the default) the control looks to its parent to determine what *BiDiMode* to use. If the control is a *TForm* object, the form uses the *BiDiMode* setting from Application. If all the *ParentBiDiMode* properties are *True*, when you change Application's *BiDiMode* property, all forms and controls in the project are updated with the new setting.

FlipChildren Method (see page 1927)	The *FlipChildren* method allows you to flip the position of a container control's children. Container controls are controls that can accept other controls, such as *TForm*, *TPanel*, and *TGroupBox*.*FlipChildren* has a single boolean parameter, *AllLevels*. When *False*, only the immediate children of the container control are flipped. When *True*, all the levels of children in the container control are flipped. Delphi flips the controls by changing the Left property and the alignment of the control. If a control's left side is five pixels from the left edge of its parent control, after it is flipped... more (see page 1927)
Additional Methods (see page 1928)	There are several other methods useful for developing applications for bi-directional users. ***VCL methods that support BiDi***
Locale-specific Features (see page 1928)	You can add extra features to your application for specific locales. In particular, for Asian language environments, you may want your application to control the input method editor (IME) that is used to convert the keystrokes typed by the user into character strings. Controls offer support in programming the IME. Most windowed controls that work directly with text input have an ImeName property that allows you to specify a particular IME that should be used when the control has input focus. They also provide an ImeMode property that specifies how the IME should convert keyboard input. ImeName introduces several... more (see page 1928)
Localization Considerations in UI Design (see page 1929)	When creating an application for several foreign markets, it is important to design your user interface so that it can accommodate the changes that occur during translation. The following topics are discussed in this section: • Text (see page 1929) • Graphic images (see page 1930) • Formats and sort order (see page 1930) • Keyboard mappings (see page 1931)
Text (see page 1929)	All text that appears in the user interface must be translated. English text is almost always shorter than its translations. Design the elements of your user interface that display text so that there is room for the text strings to grow. Create dialogs, menus, status bars, and other user interface elements that display text so that they can easily display longer strings. Avoid abbreviations—they do not exist in languages that use ideographic characters. Short strings tend to grow in translation more than long phrases. The following table provides a rough estimate of how much expansion you should plan for given... more (see page 1929)
Graphic Images (see page 1930)	Ideally, you will want to use images that do not require translation. Most obviously, this means that graphic images should not include text, which will always require translation. If you must include text in your images, it is a good idea to use a label object with a transparent background over an image rather than including the text as part of the image. There are other considerations when creating graphic images. Try to avoid images that are specific to a particular culture. For example, mailboxes in different countries look very different from each other. Religious symbols are not appropriate if... more (see page 1930)
Formats and Sort Order (see page 1930)	The date, time, number, and currency formats used in your application should be localized for the target locale. If you use only the Windows formats, there is no need to translate formats, as these are taken from the user's Windows Registry. However, if you specify any of your own format strings, be sure to declare them as resourced constants so that they can be localized. The order in which strings are sorted also varies from country to country. Many European languages include diacritical marks that are sorted differently, depending on the locale. In addition, in some countries, two-character... more (see page 1930)
Keyboard Mappings (see page 1931)	Be careful with key-combinations shortcut assignments. Not all the characters available on the US keyboard are easily reproduced on all international keyboards. Where possible, use number keys and function keys for shortcuts, as these are available on virtually all keyboards.
Isolating Resources (see page 1931)	The most obvious task in localizing an application is translating the strings that appear in the user interface. To create an application that can be translated without altering code everywhere, the strings in the user interface should be isolated into a single module. Delphi automatically creates a .dfm file that contains the resources for your menus, dialogs, and bitmaps. In addition to these obvious user interface elements, you will need to isolate any strings, such as error messages, that you present to the user. String resources are not included in the form file. You can isolate them by declaring constants... more (see page 1931)

Creating Resource DLLs (see page 1931)	Isolating resources simplifies the translation process. The next level of resource separation is the creation of a resource DLL. A resource DLL contains all the resources and only the resources for a program. Resource DLLs allow you to create a program that supports many translations simply by swapping the resource DLL. Use the Resource DLL wizard to create a resource DLL for your program. The Resource DLL wizard requires an open, saved, compiled project. It will create an RC file that contains the string tables from used RC files and **resourcestring** strings of the project, and generate a project for... more (see page 1931)
Using Resource DLLs (see page 1933)	The executable, DLLs, and packages (bpls) that make up your application contain all the necessary resources. However, to replace those resources by localized versions, you need only ship your application with localized resource DLLs that have the same name as your executable, DLL, or package files. When your application starts up, it checks the locale of the local system. If it finds any resource DLLs with the same name as the EXE, DLL, or BPL files it is using, it checks the extension on those DLLs. If the extension of the resource module matches the language and country of the... more (see page 1933)
Dynamic Switching of Resource DLLs (see page 1934)	In addition to locating a resource DLL at application startup, it is possible to switch resource DLLs dynamically at runtime. To add this functionality to your own applications, you need to include the ReInit unit in your **uses** statement. (ReInit is located in the Richedit sample in the Demos directory.) To switch languages, you should call *LoadResourceModule*, passing the LCID for the new language, and then call *ReinitializeForms*. For example, the following code switches the interface language to French:
Localizing Applications (see page 1934)	Once your application is internationalized, you can create localized versions for the different foreign markets in which you want to distribute it.

3.2.4.2.1 Creating International Applications: Overview

This topic discusses guidelines for writing applications that you plan to distribute to an international market. By planning ahead, you can reduce the amount of time and code necessary to make your application function in its foreign market as well as in its domestic market.

The following topics are discussed in this section:

- Internationalization and localization (see page 1923)
- Internationalizing applications (see page 1924)
- Localizing applications (see page 1934)

See Also

Deploying Applications (see page 1947)

3.2.4.2.2 Internationalization and Localization

To create an application that you can distribute to foreign markets, there are two major steps that need to be performed:

- Internationalization (see page 1923)
- Localization (see page 1924)

If your edition includes the Translation Tools, you can use the them to manage localization.

See Also

Internationalizing Applications (see page 1924)

Localizing Applications (see page 1934)

3.2.4.2.3 Internationalization

Internationalization is the process of enabling your program to work in multiple locales. A locale is the user's environment, which includes the cultural conventions of the target country as well as the language. Windows supports many locales, each of which is

described by a language and country pair.

See Also

Localization (see page 1924)

Internationalizing Applications (see page 1924)

3.2.4.2.4 Localization

Localization is the process of translating an application so that it functions in a specific locale. In addition to translating the user interface, localization may include functionality customization. For example, a financial application may be modified for the tax laws in different countries.

See Also

Internationalization (see page 1923)

Localizing Applications (see page 1934)

3.2.4.2.5 Internationalizing Applications

You need to complete the following steps to create internationalized applications:

- Enable your code (see page 1924) to handle strings from international character sets.
- Design your user interface (see page 1929) to accommodate the changes that result from localization.
- Isolate all resources (see page 1931) that need to be localized.

See Also

Internationalization (see page 1923)

Localizing Applications (see page 1934)

3.2.4.2.6 Enabling Application Code

You must make sure that the code in your application can handle the strings it will encounter in the various target locales. To do this, you must consider the following:

- Character sets (see page 1924)
- OEM and ANSI character sets (see page 1925)
- Multibyte character sets (see page 1925)
- Wide characters
- Locale-specific features (see page 1928)

See Also

Designing the User Interface (see page 1929)

Isolating Resources (see page 1931)

3.2.4.2.7 Character Sets

The Western editions (including English, French, and German) of Windows use the ANSI Latin-1 (1252) character set. However, other editions of Windows use different character sets. For example, the Japanese version of Windows uses the Shift-JIS character set (code page 932), which represents Japanese characters as multibyte character codes.

There are generally three types of characters sets:

- Single-byte
- Multibyte
- Wide characters

Windows and Linux both support single-byte and multibyte character sets as well as Unicode. With a single-byte character set, each byte in a string represents one character. The ANSI character set used by many western operating systems is a single-byte character set.

In a multibyte character set, some characters are represented by one byte and others by more than one byte. The first byte of a multibyte character is called the lead byte. In general, the lower 128 characters of a multibyte character set map to the 7-bit ASCII characters, and any byte whose ordinal value is greater than 127 is the lead byte of a multibyte character. Only single-byte characters can contain the null value (#0). Multibyte character sets—especially double-byte character sets (DBCS)—are widely used for Asian languages.

See Also

OEM and ANSI Character Sets (see page 1925)

Multibyte Character Sets (see page 1925)

Locale-specific Features (see page 1928)

Wide Characters

Unicode in RAD Studio

Enabling Unicode in Your Applications

3.2.4.2.8 OEM and ANSI Character Sets

It is sometimes necessary to convert between the Windows character set (ANSI) and the character set specified by the code page of the user's machine (called the OEM character set).

See Also

Character Sets (see page 1924)

Multibyte Character Sets (see page 1925)

Locale-specific Features (see page 1928)

Wide Characters

3.2.4.2.9 Multibyte Character Sets (MBCS)

The ideographic character sets used in Asia cannot use the simple 1:1 mapping between characters in the language and the one byte (8-bit) **AnsiChar** type. These languages have too many characters to be represented using the single-byte **AnsiChar**. Instead, a multibyte character set string can contain one or more bytes per character. A multibyte character set provides a way to encode characters outside the standard ANSI range into single byte strings of **AnsiChar**.

The lead byte of every multibyte character code is taken from a reserved range that depends on the specific character set. The second and subsequent bytes can sometimes be the same as the character code for a separate one-byte character, or it can fall in the range reserved for the first byte of multibyte characters. Thus, the only way to tell whether a particular byte in a string represents a single character or is part of a multibyte character is to read the string, starting at the beginning, parsing it into two or more byte characters when a lead byte from the reserved range is encountered.

When writing code for Asian locales, you must be sure to handle all string manipulation using functions that are enabled to parse strings into multibyte characters.

For these reasons, you cannot process multibyte character strings (MBCS) as you process single byte character strings. You should use a string type appropriate for MCBS data, such as **AnsiString** or a short string.

Delphi provides you with many of these runtime library functions, as listed in the following table:

Runtime library functions

AdjustLineBreaks	AnsiStrLower	ExtractFileDir
AnsiCompareFileName	AnsiStrPos	ExtractFileExt
AnsiExtractQuotedStr	AnsiStrRScan	ExtractFileName
AnsiLastChar	AnsiStrScan	ExtractFilePath
AnsiLowerCase	AnsiStrUpper	ExtractRelativePath
AnsiLowerCaseFileName	AnsiUpperCase	FileSearch
AnsiPos	AnsiUpperCaseFileName	IsDelimiter
AnsiQuotedStr	ByteToCharIndex	IsPathDelimiter
AnsiStrComp	ByteToCharLen	LastDelimiter
AnsiStrIComp	ByteType	StrByteType
AnsiStrLastChar	ChangeFileExt	StringReplace
AnsiStrLComp	CharToByteIndex	WrapText
AnsiStrLIComp	CharToByteLen	

Remember that the length of the strings in bytes does not necessarily correspond to the length of the string in characters. Be careful not to truncate strings by cutting a multibyte character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character can't be known up front. Instead, always pass a pointer to a character or a string.

Ideographic character sets can also be represented in Unicode with the **UnicodeString** or **WideString** types. The **WideString** character type is essentially the same as a Windows BSTR. **WideString** is still appropriate for use in COM applications. However, **WideString** is not reference counted, and **UnicodeString** is more flexible and efficient in other types of applications. In addition, more functions are available for handling **UnicodeString** types than **WideString**, so **UnicodeString** is generally preferred.

See Also

Character Sets (see page 1924)

OEM and ANSI Character Sets (see page 1925)

Locale-specific Features (see page 1928)

Wide Characters

Commonly Used Long String Routines (see page 2118)

Enabling Unicode in Your Applications

Unicode in RAD Studio

3.2.4.2.10 Including Bi-directional Functionality in Applications

Some languages do not follow the left to right reading order commonly found in western languages, but rather read words right to left and numbers left to right. These languages are termed bi-directional (BiDi) because of this separation. The most common bi-directional languages are Arabic and Hebrew, although other Middle East languages are also bi-directional.

TApplication has two properties, *BiDiKeyboard* and *NonBiDiKeyboard*, that allow you to specify the keyboard layout. In addition, the VCL supports bi-directional localization through the *BiDiMode* and *ParentBiDiMode* properties.

Note: Bi-directional properties are not available for cross-platform applications.

The BiDiMode property controls the reading order for the text, the placement of the vertical scrollbar, and whether the alignment is changed. Controls that have a text property, such as *Name*, display the *BiDiMode* property on the **Object Inspector**.

The *BiDiMode* property is a new enumerated type, *TBiDiMode*, with four states: *bdLeftToRight*, *bdRightToLeft*, *bdRightToLeftNoAlign*, and *bdRightToLeftReadingOnly*.

bdLeftToRight

bdLeftToRight draws text using left to right reading order. The alignment and scroll bars are not changed. For instance, when entering right to left text, such as Arabic or Hebrew, the cursor goes into push mode and the text is entered right to left. Latin text, such as English or French, is entered left to right. *bdLeftToRight* is the default value.

bdRightToLeft

bdRightToLeft draws text using right to left reading order, the alignment is changed and the scroll bar is moved. Text is entered as normal for right-to-left languages such as Arabic or Hebrew. When the keyboard is changed to a Latin language, the cursor goes into push mode and the text is entered left to right.

bdRightToLeftNoAlign

bdRightToLeftNoAlign draws text using right to left reading order, the alignment is not changed, and the scroll bar is moved.

bdRightToLeftReadingOnly

bdRightToLeftReadingOnly draws text using right to left reading order, and the alignment and scroll bars are not changed.

See Also

Localization (see page 1924)

Internationalizing Applications (see page 1924)

3.2.4.2.11 ParentBiDiMode Property

ParentBiDiMode is a Boolean property. When *True* (the default) the control looks to its parent to determine what *BiDiMode* to use. If the control is a *TForm* object, the form uses the *BiDiMode* setting from Application. If all the *ParentBiDiMode* properties are *True*, when you change Application's *BiDiMode* property, all forms and controls in the project are updated with the new setting.

See Also

Localization (see page 1924)

Internationalizing Applications (see page 1924)

3.2.4.2.12 FlipChildren Method

The *FlipChildren* method allows you to flip the position of a container control's children. Container controls are controls that can accept other controls, such as *TForm*, *TPanel*, and *TGroupBox*.*FlipChildren* has a single boolean parameter, *AllLevels*. When *False*, only the immediate children of the container control are flipped. When *True*, all the levels of children in the container control are flipped.

Delphi flips the controls by changing the Left property and the alignment of the control. If a control's left side is five pixels from the left edge of its parent control, after it is flipped the edit control's right side is five pixels from the right edge of the parent control. If the edit control is left aligned, calling *FlipChildren* will make the control right aligned.

To flip a control at design-time select **Edit ▸ Flip Children** and select either All or Selected, depending on whether you want to flip all the controls, or just the children of the selected control. You can also flip a control by selecting the control on the form, right-clicking, and selecting Flip Children from the context menu.

Note: Selecting an edit control and issuing a Flip Children|Selected command does nothing. This is because edit controls are not containers.

See Also

Localization (see page 1924)

Internationalizing Applications (see page 1924)

Additional Methods (see page 1928)

3.2.4.2.13 Additional Methods

There are several other methods useful for developing applications for bi-directional users.

VCL methods that support BiDi

Method	Description
OkToChangeFieldAlignment	Used with database controls. Checks to see if the alignment of a control can be changed.
DBUseRightToLeftAlignment	A wrapper for database controls for checking alignment.
ChangeBiDiModeAlignment	Changes the alignment parameter passed to it. No check is done for *BiDiMode* setting, it just converts left alignment into right alignment and vice versa, leaving center-aligned controls alone.
IsRightToLeft	Returns *True* if any of the right to left options are selected. If it returns *False* the control is in left to right mode.
UseRightToLeftReading	Returns *True* if the control is using right to left reading.
UseRightToLeftAlignment	Returns *True* if the control is using right to left alignment. It can be overridden for customization.
UseRightToLeftScrollBar	Returns *True* if the control is using a left scroll bar.
DrawTextBiDiModeFlags	Returns the correct draw text flags for the *BiDiMode* of the control.
DrawTextBiDiModeFlagsReadingOnly	Returns the correct draw text flags for the *BiDiMode* of the control, limiting the flag to its reading order.
AddBiDiModeExStyle	Adds the appropriate *ExStyle* flags to the control that is being created.

See Also

Localization (see page 1924)

Internationalizing Applications (see page 1924)

FlipChildren Method (see page 1927)

3.2.4.2.14 Locale-specific Features

You can add extra features to your application for specific locales. In particular, for Asian language environments, you may want your application to control the input method editor (IME) that is used to convert the keystrokes typed by the user into character strings.

Controls offer support in programming the IME. Most windowed controls that work directly with text input have an ImeName

property that allows you to specify a particular IME that should be used when the control has input focus. They also provide an ImeMode property that specifies how the IME should convert keyboard input. ImeName introduces several protected methods that you can use to control the IME from classes you define. In addition, the global Screen variable provides information about the IMEs available on the user's system.

The global *Screen* variable also provides information about the keyboard mapping installed on the user's system. You can use this to obtain locale-specific information about the environment in which your application is running.

The IME is available in VCL applications only.

See Also

Character Sets (see page 1924)

OEM and ANSI Character Sets (see page 1925)

Wide Characters

Multibyte Character Sets (see page 1925)

3.2.4.2.15 Localization Considerations in UI Design

When creating an application for several foreign markets, it is important to design your user interface so that it can accommodate the changes that occur during translation.

The following topics are discussed in this section:

- Text (see page 1929)
- Graphic images (see page 1930)
- Formats and sort order (see page 1930)
- Keyboard mappings (see page 1931)

See Also

Enabling Application Code (see page 1924)

Isolating Resources (see page 1931)

Programming for Varying Host Environments (see page 1953)

3.2.4.2.16 Text

All text that appears in the user interface must be translated. English text is almost always shorter than its translations. Design the elements of your user interface that display text so that there is room for the text strings to grow. Create dialogs, menus, status bars, and other user interface elements that display text so that they can easily display longer strings. Avoid abbreviations—they do not exist in languages that use ideographic characters.

Short strings tend to grow in translation more than long phrases. The following table provides a rough estimate of how much expansion you should plan for given the length of your English strings:

Estimating string lengths

Length of English String (in characters)	Expected Increase
1-5	100%
6-12	80%
13-20	60%

21-30	40%
31-50	20%
over 50	10%

See Also

Graphic Images (see page 1930)

Formats and Sort Order (see page 1930)

Keyboard Mappings (see page 1931)

Working with String Lists (see page 2140)

3.2.4.2.17 Graphic Images

Ideally, you will want to use images that do not require translation. Most obviously, this means that graphic images should not include text, which will always require translation. If you must include text in your images, it is a good idea to use a label object with a transparent background over an image rather than including the text as part of the image.

There are other considerations when creating graphic images. Try to avoid images that are specific to a particular culture. For example, mailboxes in different countries look very different from each other. Religious symbols are not appropriate if your application is intended for countries that have different dominant religions. Even color can have different symbolic connotations in different cultures.

See Also

Text (see page 1929)

Formats and Sort Order (see page 1930)

Keyboard Mappings (see page 1931)

Adding Graphics to Controls (see page 2154)

Working With Graphics And Multi Media (see page 2208)

3.2.4.2.18 Formats and Sort Order

The date, time, number, and currency formats used in your application should be localized for the target locale. If you use only the Windows formats, there is no need to translate formats, as these are taken from the user's Windows Registry. However, if you specify any of your own format strings, be sure to declare them as resourced constants so that they can be localized.

The order in which strings are sorted also varies from country to country. Many European languages include diacritical marks that are sorted differently, depending on the locale. In addition, in some countries, two-character combinations are treated as a single character in the sort order. For example, in Spanish, the combination *ch* is sorted like a single unique letter between *c* and *d*. Sometimes a single character is sorted as if it were two separate characters, such as the German *eszett*.

See Also

Text (see page 1929)

Graphic Images (see page 1930)

Keyboard Mappings (see page 1931)

Character Sets (see page 1924)

3.2.4.2.19 Keyboard Mappings

Be careful with key-combinations shortcut assignments. Not all the characters available on the US keyboard are easily reproduced on all international keyboards. Where possible, use number keys and function keys for shortcuts, as these are available on virtually all keyboards.

See Also

Text (see page 1929)

Graphic Images (see page 1930)

Formats and Sort Order (see page 1930)

Character Sets (see page 1924)

3.2.4.2.20 Isolating Resources

The most obvious task in localizing an application is translating the strings that appear in the user interface. To create an application that can be translated without altering code everywhere, the strings in the user interface should be isolated into a single module. Delphi automatically creates a .dfm file that contains the resources for your menus, dialogs, and bitmaps.

In addition to these obvious user interface elements, you will need to isolate any strings, such as error messages, that you present to the user. String resources are not included in the form file. You can isolate them by declaring constants for them using the **resourcestring** keyword. For more information about resource string constants, see the *Delphi Language Guide*. It is best to include all resource strings in a single, separate unit.

For information on using resource DLLs in your applications see Creating Resource DLLs (see page 1931) and Using Resource DLLs (see page 1933).

See Also

Enabling Application Code (see page 1924)

Designing the User Interface (see page 1929)

Using Forms (see page 1990)

Creating Resource DLLs (see page 1931)

3.2.4.2.21 Creating Resource DLLs

Isolating resources simplifies the translation process. The next level of resource separation is the creation of a resource DLL. A resource DLL contains all the resources and only the resources for a program. Resource DLLs allow you to create a program that supports many translations simply by swapping the resource DLL.

Use the Resource DLL wizard to create a resource DLL for your program. The Resource DLL wizard requires an open, saved, compiled project. It will create an RC file that contains the string tables from used RC files and **resourcestring** strings of the project, and generate a project for a resource only DLL that contains the relevant forms and the created RES file. The RES file is compiled from the new RC file.

You should create a resource DLL for each translation you want to support. Each resource DLL should have a file name extension specific to the target locale. The first two characters indicate the target language, and the third character indicates the country of the locale. If you use the Resource DLL wizard, this is handled for you. Otherwise, use the following code to obtain the locale code for the target translation:

```
unit locales;
```

```
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
type
  TForm1 = class(TForm)
    Button1: TButton;
    LocaleList: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
function GetLocaleData(ID: LCID; Flag: DWORD): string;
var
  BufSize: Integer;
begin
  BufSize := GetLocaleInfo(ID, Flag, nil, 0);
  SetLength(Result, BufSize);
  GetLocaleinfo(ID, Flag, PChar(Result), BufSize);
  SetLength(Result, BufSize - 1);
end;
{ Called for each supported locale. }
function LocalesCallback(Name: PChar): Bool; stdcall;
var
  LCID: Integer;
begin
  LCID := StrToInt('$' + Copy(Name, 5, 4));
  Form1.LocaleList.Items.Add(GetLocaleData(LCID, LOCALE_SLANGUAGE));
  Result := Bool(1);
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  with Languages do
  begin
    for I := 0 to Count - 1 do
    begin
      ListBox1.Items.Add(Name[I]);
    end;
  end;
end;
/* This callback fills a listbox with the strings and their associated languages and
countries*/
BOOL __stdcall EnumLocalesProc(char* lpLocaleString)
{
AnsiString LocaleName, LanguageName, CountryName;
LCID lcid;
lcid = StrToInt("$" + AnsiString(lpLocaleString));
LocaleName = GetLocaleStr(lcid, LOCALE_SABBREVLANGNAME, "");
LanguageName = GetLocaleStr(lcid, LOCALE_SNATIVELANGNAME, "");
CountryName = GetLocaleStr(lcid, LOCALE_SNATIVECTRYNAME, "");
if (lstrlen(LocaleName.c_str()) > 0)
Form1->ListBox1->Items->Add(LocaleName + ":" + LanguageName + "-" + CountryName);
return TRUE;
}
/* This call causes the callback to execute for every locale */
EnumSystemLocales((LOCALE_ENUMPROC)EnumLocalesProc, LCID_SUPPORTED);
```

See Also

Using Resource DLLs (see page 1933)

3.2.4.2.22 Using Resource DLLs

The executable, DLLs, and packages (bpls) that make up your application contain all the necessary resources. However, to replace those resources by localized versions, you need only ship your application with localized resource DLLs that have the same name as your executable, DLL, or package files.

When your application starts up, it checks the locale of the local system. If it finds any resource DLLs with the same name as the EXE, DLL, or BPL files it is using, it checks the extension on those DLLs. If the extension of the resource module matches the language and country of the system locale, your application will use the resources in that resource module instead of the resources in the executable, DLL, or package. If there is not a resource module that matches both the language and the country, your application will try to locate a resource module that matches just the language. If there is no resource module that matches the language, your application will use the resources compiled with the executable, DLL, or package.

If you want your application to use a different resource module than the one that matches the locale of the local system, you can set a locale override entry in the Windows registry. Under the HKEY_CURRENT_USER\Software\Borland\Locales key, add your application's path and file name as a string value and set the data value to the extension of your resource DLLs. At startup, the application will look for resource DLLs with this extension before trying the system locale. Setting this registry entry allows you to test localized versions of your application without changing the locale on your system.

For example, the following procedure can be used in an install or setup program to set the registry key value that indicates the locale to use when loading applications:

```
procedure SetLocalOverrides(FileName: string, LocaleOverride: string);
var
Reg: TRegistry;
begin
Reg := TRegistry.Create;
try
if Reg.OpenKey('Software\Borland\Locales', True) then
Reg.WriteString(LocalOverride, FileName);
finally
Reg.Free;
  end;
end;
void SetLocalOverrides(char* FileName, char* LocaleOverride)
{
HKEY Key;
const char* LocaleOverrideKey = "Software\\Borland\\Locales";
if (RegOpenKeyEx(HKEY_CURRENT_USER, LocaleOverrideKey, 0, KEY_ALL_ACCESS, &Key)
== ERROR_SUCCESS) {
if (lstrlen(LocaleOverride) == 3)
RegSetValueEx(Key, FileName, 0, REG_SZ, (const BYTE*)LocaleOverride, 4);
RegCloseKey(Key);
}
}
```

Within your application, use the global *FindResourceHInstance* function to obtain the handle of the current resource module. For example:

```
LoadStr(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, SizeOf(szQuery));
LoadString(FindResourceHInstance(HInstance), IDS_AmountDueName, szQuery, sizeof(szQuery));
```

You can ship a single application that adapts itself automatically to the locale of the system it is running on, simply by providing the appropriate resource DLLs.

See Also

Creating Resource DLLs (see page 1931)

3.2.4.2.23 Dynamic Switching of Resource DLLs

In addition to locating a resource DLL at application startup, it is possible to switch resource DLLs dynamically at runtime. To add this functionality to your own applications, you need to include the ReInit unit in your **uses** statement. (ReInit is located in the Richedit sample in the Demos directory.) To switch languages, you should call *LoadResourceModule*, passing the LCID for the new language, and then call *ReinitializeForms*.

For example, the following code switches the interface language to French:

```
const
  FRENCH = (SUBLANG_FRENCH shl 10) or LANG_FRENCH;
if LoadNewResourceModule(FRENCH) <> 0 then
  ReinitializeForms;
const  FRENCH = MAKELANGID(SUBLANG_FRENCH, LANG_FRENCH);
if (LoadNewResourceModule(FRENCH))
  ReinitializeForms();
```

The advantage of this technique is that the current instance of the application and all of its forms are used. It is not necessary to update the registry settings and restart the application or re-acquire resources required by the application, such as logging in to database servers.

When you switch resource DLLs the properties specified in the new DLL overwrite the properties in the running instances of the forms.

Note: Any changes made to the form properties at runtime will be lost. Once the new DLL is loaded, default values are not reset. Avoid code that assumes that the form objects are reinitialized to the their startup state, apart from differences due to localization.

See Also

Creating Resource DLLs (see page 1931)

3.2.4.2.24 Localizing Applications

Once your application is internationalized, you can create localized versions for the different foreign markets in which you want to distribute it.

Localizing resources

Ideally, your resources have been isolated into a resource DLL that contains form files (.dfm in VCL applications) and a resource file. You can open your forms in the IDE and translate the relevant properties.

Note: In a resource DLL project, you cannot add or delete components. It is possible, however, to change properties in ways that could cause runtime errors, so be careful to modify only those properties that require translation. To avoid mistakes, you can configure the Object Inspector

to display only Localizable properties; to do so, right-click in the **Object Inspector** and use the View menu to filter out unwanted property categories. You can open the RC file and translate relevant strings. Use the StringTable editor by opening the RC file from the **Project Manager**.

See Also

Localization (see page 1924)

Internationalizing Applications (see page 1924)

3.2.4.3 Delphi programming fundamentals

Topics

Name	Description
Designing Applications (see page 1935)	You can design any kind of 32-bit applicationfrom general—purpose utilities to sophisticated data access programs or distributed applications. As you visually design the user interface for your application, the Form Designer generates the underlying Delphi code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment. You can create your own components using the... more (see page 1935)
Creating Projects (see page 1936)	All application development revolves around projects. When you create an application in Delphi you are creating a project. A project is a collection of files that make up an application. Some of these files are created at design time. Others are generated automatically when you compile the project source code. You can view the contents of a project in a project management tool called the **Project Manager**. The **Project Manager** lists, in a hierarchical view, the unit names, the forms contained in the unit (if there is one), and shows the paths to the files in the project. Although... more (see page 1936)
Editing Code (see page 1936)	The Code editor is a full-featured ASCII editor. If using the visual programming environment, a form is automatically displayed as part of a new project. You can start designing your application interface by placing objects on the form and modifying how they work in the **Object Inspector**. But other programming tasks, such as writing event handlers for objects, must be done by typing the code. The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code... more (see page 1936)
Compiling Applications (see page 1937)	When you have finished designing your application interface on the form and writing additional code so it does what you want, you can compile the project from the IDE or from the command line. All projects have as a target a single distributable executable file. You can view or test your application at various stages of development by compiling, building, or running it: • When you compile, only units that have changed since the last compile are recompiled. • When you build, all units in the project are compiled, regardless of whether they have changed since the last compile. This technique is... more (see page 1937)
Debugging Applications (see page 1937)	With the integrated debugger, you can find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging. The integrated debugger can track down both runtime errors and logic errors. By running to specific program locations and viewing the variable values, the functions on the call stack, and the program output, you can monitor how your program behaves and find the areas where it is not behaving as designed. You can also use exception handling to recognize, locate, and deal with errors. Exceptions... more (see page 1937)
Deploying Applications (see page 1937)	Delphi includes add-on tools to help with application deployment. For example, InstallAware Express (not available in all editions) helps you to create an installation package for your application that includes all of the files needed for running a distributed application. TeamSource software (not available in all editions) is also available for tracking application updates. **Note:** Not all editions have deployment capabilities. Refer to Deploying Applications (see page 1947) for specific information on deployment.

3.2.4.3.1 Designing Applications

You can design any kind of 32-bit applicationfrom general—purpose utilities to sophisticated data access programs or distributed applications.

As you visually design the user interface for your application, the Form Designer generates the underlying Delphi code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment.

You can create your own components using the Delphi language. Most of the components provided are written in Delphi. You can add components that you write to the **Tool palette** and customize the palette for your use by including new tabs if needed.

Creating applications (see page 1883) introduces support for different types of applications.

See Also

Developing the Application User Interface: Overview (see page 1983)

Working with Components (see page 2142)

Creating and Managing Menus (see page 1975)

3.2.4.3.2 Creating Projects

All application development revolves around projects. When you create an application in Delphi you are creating a project. A project is a collection of files that make up an application. Some of these files are created at design time. Others are generated automatically when you compile the project source code.

You can view the contents of a project in a project management tool called the **Project Manager**. The **Project Manager** lists, in a hierarchical view, the unit names, the forms contained in the unit (if there is one), and shows the paths to the files in the project. Although you can edit many of these files directly, it is often easier and more reliable to use the visual tools.

At the top of the project hierarchy is a group file. You can combine multiple projects into a project group. This allows you to open more than one project at a time in the **Project Manager**. Project groups let you organize and work on related projects, such as applications that function together or parts of a multi-tiered application. If you are only working on one project, you do not need a project group file to create an application.

Project files, which describe individual projects, files, and associated options, have a .dpr extension. Project files contain directions for building an application or shared object. When you add and remove files using the **Project Manager**, the project file is updated. You specify project options using a Project Options dialog which has tabs for various aspects of your project such as forms, application, and compiler. These project options are stored in the project file with the project.

Units and forms are the basic building blocks of an application. A project can share any existing form and unit file including those that reside outside the project directory tree. This includes custom procedures and functions that have been written as standalone routines.

If you add a shared file to a project, realize that the file is not copied into the current project directory; it remains in its current location. Adding the shared file to the current project registers the file name and path in the **uses** clause of the project file. Delphi automatically handles this as you add units to a project.

When you compile a project, it does not matter where the files that make up the project reside. The compiler treats shared files the same as those created by the project itself.

See Also

Developing the Application User Interface: Overview (see page 1983)

3.2.4.3.3 Editing Code

The Code editor is a full-featured ASCII editor. If using the visual programming environment, a form is automatically displayed as part of a new project. You can start designing your application interface by placing objects on the form and modifying how they

work in the **Object Inspector**. But other programming tasks, such as writing event handlers for objects, must be done by typing the code.

The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code editor and add more components within the editor by typing code. As you type code into the editor, the compiler is constantly scanning for changes and updating the form with the new layout. You can then go back to the form, view and test the changes you made in the editor, and continue adjusting the form from there.

The code generation and property streaming systems are completely open to inspection. The source code for everything that is included in your final executable file—all of the VCL objects, RTL sources, and project files—can be viewed and edited in the Code editor.

3.2.4.3.4 Compiling Applications

When you have finished designing your application interface on the form and writing additional code so it does what you want, you can compile the project from the IDE or from the command line.

All projects have as a target a single distributable executable file. You can view or test your application at various stages of development by compiling, building, or running it:

- When you compile, only units that have changed since the last compile are recompiled.
- When you build, all units in the project are compiled, regardless of whether they have changed since the last compile. This technique is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized. It's also important to build when you"ve changed global compiler directives to ensure that all code compiles in the proper state. You can also test the validity of your source code without attempting to compile the project.
- When you run, you compile and then execute your application. If you modified the source code since the last compilation, the compiler recompiles the changed files and any files that depend on them.

If you have grouped several projects together, you can compile or build all projects in a single project group at once. Choose **Project** ▸ **Compile All Projects**, or **Project** ▸ **Build All Projects** with the project group selected in the **Project Manager**.

3.2.4.3.5 Debugging Applications

With the integrated debugger, you can find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging.

The integrated debugger can track down both runtime errors and logic errors. By running to specific program locations and viewing the variable values, the functions on the call stack, and the program output, you can monitor how your program behaves and find the areas where it is not behaving as designed.

You can also use exception handling to recognize, locate, and deal with errors. Exceptions are classes, like other classes in Delphi, except, by convention, they begin with an initial E rather than a T.

3.2.4.3.6 Deploying Applications

Delphi includes add-on tools to help with application deployment. For example, InstallAware Express (not available in all editions) helps you to create an installation package for your application that includes all of the files needed for running a distributed application. TeamSource software (not available in all editions) is also available for tracking application updates.

Note: Not all editions have deployment capabilities.

Refer to Deploying Applications (see page 1947) for specific information on deployment.

3.2.4.4 Deploying applications

Topics

Name	Description
Deploying BDE Applications (see page 1941)	**Warning:** The Borland Database Engine (BDE) has been deprecated, so BDE will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress. The Borland Database Engine (BDE) defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables. Database access for an application is provided by various... more (see page 1941)
Deploying dbExpress Database Applications (see page 1942)	dbExpress is a set of thin, native drivers that provide fast access to database information.
Deploying On Apache Servers (see page 1943)	WebBroker supports Apache version 1.3.9 and later for DLLs and CGI applications. Modules and applications are enabled and configured by modifying Apache's httpd.conf file (normally located in your Apache installation's \conf directory).
Accommodating Varying Color Depths (see page 1945)	To account for all deployment computers not being configured with the same color availability, the safest way is to use graphics with the least possible number of colors. This is especially true for control glyphs, which should typically use 16-color graphics. For displaying pictures, either provide multiple copies of the images in different resolutions and color depths or explain in the application the minimum resolution and color requirements for the application.
Deploying ActiveX Controls (see page 1945)	ActiveX controls that you create need to be registered on the deployment computer before use. Installation programs such as InstallAware Express automate this registration process. To manually register an ActiveX control, choose **Run ▶ ActiveX Server** in the IDE, use the TRegSvr demo application in \Bin or use the Microsoft utility REGSRV32.EXE (not included with Windows 9x versions). DLLs that support an ActiveX control also need to be distributed with an application.
Application Files, Listed by File Name Extension (see page 1945)	The following types of files may need to be distributed with an application. ***Application files***
Borland Database Engine (see page 1946)	**Warning:** The Borland Database Engine (BDE) has been deprecated, so BDE will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress. You can use the Borland Database Engine (BDE) to provide database access for standard Delphi data components. See the BDEDEPLOY document for specific rights and limitations on redistributing the BDE (see C:\Program Files\Common Files\Borland Shared\BDE). You should use InstallAware Express (or other certified installation program) for installing the BDE. InstallAware Express creates the necessary registry... more (see page 1946)
Considerations When Dynamically Resizing Forms and Controls (see page 1946)	If the forms and visual controls for an application are dynamically resized, accommodate all aspects of the resizing process to ensure optimal appearance of the application under all possible screen resolutions. Here are some factors to consider when dynamically resizing the visual elements of an application: • The resizing of forms and visual controls is done at a ratio calculated by comparing the screen resolution of the development computer to that of the computer onto which the application installed. Use a constant to represent one dimension of the screen resolution on the development computer: either the height or the width, in... more (see page 1946)

Considerations When Not Dynamically Resizing (see page 1947)	If the forms and visual controls that make up an application are not dynamically resized at runtime, design the application's elements for the lowest resolution. Otherwise, the forms of an application run on a computer configured for a lower screen resolution than the development computer may overlap the boundaries of the screen. For example, if the development computer is set up for a screen resolution of 1024x768 and a form is designed with a width of 700 pixels, not all of that form will be visible within the desktop on a computer configured for a 640x480 screen resolution.
DLL Locations (see page 1947)	You can install DLL files used only by a single application in the same directory as the application. DLLs that will be used by a number of applications should be installed in a location accessible to all of those applications. A common convention for locating such community DLLs is to place them either in the Windows or the Windows\System directory. A better way is to create a dedicated directory for the common .DLL files, similar to the way the Borland Database Engine is installed.
Deploying Applications: Overview (see page 1947)	Once your application is up and running, you can deploy it. That is, you can make it available for others to run. A number of steps must be taken to deploy an application to another computer so that the application is completely functional. The steps required by a given application vary, depending on the type of application. The following sections describe these steps when deploying the following applications: • Deploying General Applications (see page 1949) • Deploying Database Applications (see page 1948) • Deploying Web Applications (see page 1949) • Programming for Varying Host Environments (see page 1953) • Software License Requirements (see page 1954)
Deploying Database Applications (see page 1948)	Applications that access databases involve special installation considerations beyond copying the application's executable file onto the host computer. Database access is most often handled by a separate database engine, the files of which cannot be linked into the application's executable file. The data files, when not created beforehand, must be made available to the application. Multi-tier database applications require additional handling on installation, because the files that make up the application are typically located on multiple computers. Since several different database technologies (ADO, BDE, dbExpress, and InterBase Express) are supported, deployment requirements differ for each. Regardless of which you are... more (see page 1948)
Deploying General Applications (see page 1949)	Beyond the executable file, an application may require a number of supporting files, such as DLLs, package files, and helper applications. In addition, the Windows registry may need to contain entries for an application, from specifying the location of supporting files to simple program settings. The process of copying an application's files to a computer and making any needed registry settings can be automated by an installation program, such as InstallShield Express. Nearly all types of applications include the following issues: • Using installation programs (see page 1954) • Identifying application files (see page 1950) • Helper applications (see page 1950) • DLL locations (see page 1947) Database and Web applications require additional installation steps. For... more (see page 1949)

Deploying Web Applications (see page 1949)	Some Delphi applications are designed to be run over the World Wide Web, such as those in the form of Server-side Extension DLLs (ISAPI and Apache), CGI applications, and ActiveForms. The steps for deploying Web applications are the same as those for general applications (see page 1949), except the application's files are deployed on the Web server. Here are some special considerations for deploying Web applications: • For BDE database applications, the Borland Database Engine (or alternate database engine) is installed with the application files on the Web server. • For dbExpress applications, the dbExpress DLLs must be included in the path. If... more (see page 1949)
Fonts (see page 1950)	Windows comes with a standard set of TrueType and raster fonts. Linux comes with a standard set of fonts, depending on the distribution. When designing an application to be deployed on other computers, realize that not all computers have fonts outside the standard sets. Text components used in the application should all use fonts that are likely to be available on all deployment computers. When use of a nonstandard font is absolutely necessary in an application, you need to distribute that font with the application. Either the installation program or the application itself must install the font on the deployment... more (see page 1950)
Helper Applications (see page 1950)	Helper applications are separate programs without which your application would be partially or completely unable to function. Helper applications may be those supplied with the operating system, by CodeGear, or by third-party products. An example of a helper application is the InterBase utility program Server Manager, which administers InterBase databases, users, and security. If an application depends on a helper program, be sure to deploy it with your application, where possible. Distribution of helper programs may be governed by redistribution license agreements. Consult the helper program documentation for specific information.
Identifying Application Files (see page 1950)	Besides the executable file, a number of other files may need to be distributed with an application. • Application files (see page 1945) • Package files (see page 1952) • Merge modules (see page 1951) • ActiveX controls (see page 1945)
Merge Modules (see page 1951)	InstallAware Express is based on Windows Installer (MSI) technology. With MSI-based setup tools such as InstallAware Express, you can use merge modules for deploying runtime packages. Merge modules provide a standard method that you can use to deliver shared code, files, resources, Registry entries, and setup logic to applications as a single compound file. The runtime libraries have some interdependencies because of the way they are grouped together. The result of this is that when one package is added to an install project, the install tool automatically adds or reports a dependency on one or more other packages. For... more (see page 1951)
Deploying Multi-tiered Database Applications (DataSnap) (see page 1952)	DataSnap provides multi-tier database capability to Delphi applications by allowing client applications to connect to providers in an application server. Install DataSnap along with a multi-tier application using InstallAware (or other CodeGear-certified installation scripting utility). See the DEPLOY document (found in the main Delphi directory) for details on the files that need to be redistributed with an application. Also see the REMOTE document for related information on what DataSnap files can be redistributed and how.
Package Files (see page 1952)	If the application uses runtime packages, those package files need to be distributed with the application. InstallShield Express handles the installation of package files the same as DLLs, copying the files and making necessary entries in the Windows registry. You can also use merge modules for deploying runtime packages with MSI-based setup tools including InstallShield Express. See Merge modules (see page 1951) for details. CodeGear recommends installing the runtime package files supplied by CodeGear in the Windows\System directory. This serves as a common location so that multiple applications would have access to a single instance of the files. For packages you... more (see page 1952)

Programming for Varying Host Environments (see page 1953)	Due to the nature of various operating system environments, there are a number of factors that vary with user preference or configuration. The following factors can affect an application deployed to another computer: • Screen resolutions and color depths (see page 1953) • Fonts (see page 1950) • Operating system versions (see page 1955) • Helper applications (see page 1950) • DLL locations (see page 1947)
Screen Resolutions and Color Depths (see page 1953)	The size of the desktop and number of available colors on a computer is configurable and dependent on the hardware installed. These attributes are also likely to differ on the deployment computer compared to those on the development computer. An application's appearance (window, object, and font sizes) on computers configured for different screen resolutions can be handled in various ways: • Design the application for the lowest resolution users will have (typically, 640x480). Take no special actions to dynamically resize objects to make them proportional to the host computer's screen display. Visually, objects will appear smaller the higher the resolution is... more (see page 1953)
Software License Requirements (see page 1954)	The distribution of some files associated with Delphi applications is subject to limitations or cannot be redistributed at all. The following documents describe the legal stipulations regarding the distribution of these files:
Using Installation Programs (see page 1954)	Simple applications that consist of only an executable file are easy to install on a target computer. Just copy the executable file onto the computer. However, more complex applications that comprise multiple files require more extensive installation procedures. These applications require dedicated installation programs. Setup toolkits automate the process of creating installation programs, often without needing to write any code. Installation programs created with Setup toolkits perform various tasks inherent to installing Delphi applications, including: copying the executable and supporting files to the host computer, making Windows registry entries, and installing the Borland Database Engine for BDE database applications.... more (see page 1954)
Operating System Versions (see page 1955)	When using operating system APIs or accessing areas of the operating system from an application, there is the possibility that the function, operation, or area may not be available on computers with different operating system versions. To account for this possibility, you have a few options: • Specify in the application's system requirements the versions of the operating system on which the application can run. It is the user's responsibility to install and use the application only under compatible operating system versions. • Check the version of the operating system as the application is installed. If an incompatible version of the operating... more (see page 1955)

3.2.4.4.1 Deploying BDE Applications

Warning: The Borland Database Engine (BDE) has been deprecated, so BDE will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

The Borland Database Engine (BDE) defines a large API for interacting with databases. Of all the data access mechanisms, the BDE supports the broadest range of functions and comes with the most supporting utilities. It is the best way to work with data in Paradox or dBASE tables.

Database access for an application is provided by various database engines. An application can use the BDE or a third-party database engine. Borland Database Engine (see page 1946) describes installation of the database access elements of an application.

See Also

Deploying General Applications (see page 1949)

Deploying Web Applications (see page 1949)

Programming for Varying Host Environments (see page 1953)

Software License Requirements (see page 1954)

Deploying Packages (see page 2215)

Deploying dbExpress Database Applications (see page 1942)

3.2.4.4.2 Deploying dbExpress Database Applications

dbExpress is a set of thin, native drivers that provide fast access to database information.

Driver packages

You can deploy a dbExpress application either as a standalone executable file or as an executable file that deploys associated dbExpress driver packages. However, you always need to deploy the database DLL for whatever databases you are using.

There are separate driver packages for each driver, which eliminates the need for a dbxdrivers.ini file. The TDBXDriver implementations have default settings for both driver and connection properties. These settings can be overridden by connection property settings. This allows many connection property settings to be minimized to a small number of property settings, such as database, user, and password.

Driver packages also contain driver specific metadata implementations. The DbxReadOnlyMetaData and DbxMetaData packages that contained metadata implementations for all drivers are no longer needed and have been removed from the product. Instead, these metadata implementations are included in their respective driver package. This makes deployment of dbExpress applications more compact, since only driver specific metadata needs to be deployed. Previously, metadata for all databases was included in the deployment, even when the application was built to run against only one database. If runtime packages are used, the number needed for deployment has been reduced by 2.

SqlExpr.pas no longer has **DbxDefaultDrivers** in its **uses** clause. If driver units are not added to an applications form or data module, the application cannot load the driver. The new driver specific units that must be added to a uses clause of your application are: dbxmysql, dbxInterbase, dbxdb2, dbxinformix, dbxmssql, dbxoracle, dbxsybase, and dbxclient (if using Blackfish SQL). This is only needed for statically linked applications.

This table below lists the database packages needed for deployment for non-standalone executables.

Note: All applications need the DbxCommonDriver120.bpl package if not built standalone.

dbExpress deployment with driver packages

Database package	When to Deploy
DbxCommonDriver120.bpl	All applications connecting to databases
DBXClientDriver120.bpl	Applications connecting to Blackfish SQL databases
DBXInformixDriver120.bpl	Applications connecting to Informix databases
DBXInterBaseDriver120.bpl	Applications connecting to InterBase databases
DBXOracleDriver120.bpl	Applications connecting to Oracle databases

DBXDb2Driver120.bpl	Applications connecting to DB2 databases
DBXSybaseASADriver120.bpl	Applications connecting to Adaptive Sybase Anywhere databases
DBXSybaseASEDriver120.bpl	Applications connecting to Sybase databases
DBXMSSQLDriver120.bpl	Applications connecting to MSSQL databases
DBXMySQLDriver120.bpl	Applications connecting to MySQL 5.0.27 databases

DbxClient drivers, currently used by Blackfish SQL, require DbxClientDriver.bpl to be deployed if packages are used. Otherwise the entire driver can be linked into an executable file, and no additional package or DLL needs to be deployed.

Driver DLLs

For database applications using Dynalink drivers, you cannot deploy a standalone executable. Instead, you can deploy associated dbExpress drivers and DLLs with your executable. If you are using DataSnap, you need to include the DataSnap DLL. The following table lists the appropriate DLLs and when to include them:

dbExpress deployment with driver DLLs

Database DLL	When to Deploy
No DLL needed	Applications connecting to Blackfish SQL databases
dbxinf.dll	Applications connecting to Informix databases
dbxint.dll	Applications connecting to InterBase databases
dbxora.dll	Applications connecting to Oracle databases
dbxdb2.dll	Applications connecting to DB2 databases
dbxasa.dll	Applications connecting to Adaptive Sybase Anywhere databases
dbxase.dll	Applications connecting to Sybase databases
dbxmss.dll	Applications connecting to MSSQL databases
dbxmys.dll	Applications connecting to MySQL 5.0.27 databases
Midas.dll	Required by database applications that use client datasets

See Using Unidirectional Datasets (see page 1823) for more information about using the dbExpress components.

See Also

Deploying General Applications (see page 1949)

Deploying Web Applications (see page 1949)

Programming for Varying Host Environments (see page 1953)

Software License Requirements (see page 1954)

Deploying Packages (see page 2215)

3.2.4.4.3 Deploying On Apache Servers

WebBroker supports Apache version 1.3.9 and later for DLLs and CGI applications.

Modules and applications are enabled and configured by modifying Apache's httpd.conf file (normally located in your Apache installation's \conf directory).

Enabling modules

Your DLLs should be physically located in the Apache Modules subdirectory.

Two modifications to httpd.conf are required to enable a module.

The first modification is to add a LoadModule entry to let Apache locate and load your DLL. For example:

```
LoadModule MyApache_module modules/Project1.dll
```

Replace MyApache_module with the exported module name from your DLL. To find the module name, in your project source, look for the exports line. For example:

```
exports
  apache_module name 'MyApache_module';
```

The second modification is to add a resource locator entry (may be added anywhere in httpd.conf after the LoadModule entry). For example:

```
# Sample location specification for a project named project1.
<Location /project1>
SetHandler project1-handler
</Location>
```

This allows all requests to http://www.somedomain.com/project1 to be passed on to the Apache module.

The SetHandler directive specifies the Web server application that handles the request. The SetHandler argument should be set to the value of the ContentType global variable.

CGI applications

When creating CGI applications, the physical directory (specified in the Directory directive of the httpd.conf file) must have the ExecCGI option and the SetHandler clause set to allow execution of programs so the CGI script can be executed. To ensure that permissions are set up properly, use the Alias directive with both Options ExecCGI and SetHandler enabled.

Note: An alternative approach is to use the ScriptAlias directive (without Options ExecCGI), but using this approach can prevent your CGI application from reading any files in the ScriptAlias directory.

The following httpd.conf line is an example of using the Alias directive to create a virtual directory on your server and mark the exact location of your CGI script:

```
Alias/MyWeb/"c:/httpd/docs/MyWeb/"
```

This would allow requests such as /MyWeb/mycgi.exe to be satisfied by running the script c:\httpd\docs\MyWeb\mycgi.exe.

You can also set Options to All or to ExecCGI using the Directory directive in httpd.conf. The Options directive controls which server features are available in a particular directory.

Directory directives are used to enclose a set of directives that apply to the named directory and its subdirectories. An example of the Directory directive is shown below:

```
<Directory "c:/httpd/docs/MyWeb">
AllowOverride None
Options ExecCGI
Order allow,deny
Allow from all
AddHandler cgi-script exe cgi
</Directory>
```

In this example, Options is set to ExecCGI permitting execution of CGI scripts in the directory MyWeb. The AddHandler clause lets Apache know that files with extensions such as exe and cgi are CGI scripts (executables).

Note: Apache executes locally on the server within the account specified in the User directive in the httpd.conf file. Make sure that the user has the appropriate rights to access the resources needed by the application.

See the Apache LICENSE file, included with your Apache distribution, for additional deployment information. For additional Apache configuration information, see http://www.apache.org.

See Also

Deploying General Applications (see page 1949)

Deploying Database Applications (see page 1948)

Programming for Varying Host Environments (see page 1953)

Software License Requirements (see page 1954)

Deploying Packages (see page 2215)

Creating Internet Server Applications (see page 2251)

3.2.4.4.4 Accommodating Varying Color Depths

To account for all deployment computers not being configured with the same color availability, the safest way is to use graphics with the least possible number of colors. This is especially true for control glyphs, which should typically use 16-color graphics. For displaying pictures, either provide multiple copies of the images in different resolutions and color depths or explain in the application the minimum resolution and color requirements for the application.

See Also

Considerations When Not Dynamically Resizing (see page 1947)

Considerations When Dynamically Resizing Forms and Controls (see page 1946)

3.2.4.4.5 Deploying ActiveX Controls

ActiveX controls that you create need to be registered on the deployment computer before use. Installation programs such as InstallAware Express automate this registration process. To manually register an ActiveX control, choose **Run ▸ ActiveX Server** in the IDE, use the TRegSvr demo application in \Bin or use the Microsoft utility REGSRV32.EXE (not included with Windows 9x versions).

DLLs that support an ActiveX control also need to be distributed with an application.

See Also

Application Files (see page 1945)

Using Installation Programs (see page 1954)

3.2.4.4.6 Application Files, Listed by File Name Extension

The following types of files may need to be distributed with an application.

Application files

Type	File Name Extension
Program files	.exe and .dll
Package files	.bpl and .dcp
Help files	.hlp, .cnt, and .toc (if used) or any other Help files your application supports
ActiveX files	.ocx (sometimes supported by a DLL)
Local table files	.dbf, .mdx, .dbt, .ndx, .db, .px, .y*, .x*, .mb, .val, .qbe, .gd*

See Also

Package Files (see page 1952)

ActiveX Controls (see page 1945)

3.2.4.4.7 Borland Database Engine

Warning: The Borland Database Engine (BDE) has been deprecated, so BDE will not be enhanced. For instance, BDE will never have Unicode support. You should not undertake new development with BDE. Consider migrating your existing database applications from BDE to dbExpress.

You can use the Borland Database Engine (BDE) to provide database access for standard Delphi data components. See the BDEDEPLOY document for specific rights and limitations on redistributing the BDE (see C:\Program Files\Common Files\Borland Shared\BDE).

You should use InstallAware Express (or other certified installation program) for installing the BDE. InstallAware Express creates the necessary registry entries and defines any aliases the application may require. Using a certified installation program to deploy the BDE files and subsets is important because:

- Improper installation of the BDE or BDE subsets can cause other applications using the BDE to fail. Such applications include not only CodeGear products, but many third-party programs that use the BDE.
- Under 32-bit Windows 95/NT and later, BDE configuration information is stored in the Windows registry instead of .ini files, as was the case under 16-bit Windows. Making the correct entries and deletions for install and uninstall is a complex task.

It is possible to install only as much of the BDE as an application actually needs. For instance, if an application only uses Paradox tables, it is only necessary to install that portion of the BDE required to access Paradox tables. This reduces the disk space needed for an application. Certified installation programs, like InstallAware Express, are capable of performing partial BDE installations. Be sure to leave BDE system files that are not used by the deployed application, but that are needed by other programs.

See Also

dbExpress Components (see page 16)

3.2.4.4.8 Considerations When Dynamically Resizing Forms and Controls

If the forms and visual controls for an application are dynamically resized, accommodate all aspects of the resizing process to ensure optimal appearance of the application under all possible screen resolutions. Here are some factors to consider when dynamically resizing the visual elements of an application:

- The resizing of forms and visual controls is done at a ratio calculated by comparing the screen resolution of the development computer to that of the computer onto which the application installed. Use a constant to represent one dimension of the screen resolution on the development computer: either the height or the width, in pixels. Retrieve the same dimension for the user's computer at runtime using the *TScreen.*Height or *TScreen*.Width property. Divide the value for the development computer by the value for the user's computer to derive the difference ratio between the two computers' screen resolutions.
- Resize the visual elements of the application (forms and controls) by reducing or increasing the size of the elements and their positions on forms. This resizing is proportional to the difference between the screen resolutions on the development and user computers. Resize and reposition visual controls on forms automatically by setting the *CustomForm*.*Scaled* form's Scaled property to *True* and calling *TWinControl.ScaleBy* its ScaleBy method . The *ScaleBy* method does not change the form's height and width, though. Do this manually by multiplying the current values for the *Height* and *Width* properties by the screen resolution difference ratio.
- The controls on a form can be resized manually, instead of automatically with the *TWinControl*.*ScaleBy* method , by referencing each visual control in a loop and setting its dimensions and position. The *Height* and *Width* property values for visual controls are multiplied by the screen resolution difference ratio. Reposition visual controls proportional to screen resolution differences by multiplying the *Top* and *Left* property values by the same ratio.
- If an application is designed on a computer configured for a higher screen resolution than that on the user's computer, font

sizes will be reduced in the process of proportionally resizing visual controls. If the size of the font at design time is too small, the font as resized at runtime may be unreadable. For example, the default font size for a form is 8. If the development computer has a screen resolution of 1024x768 and the user's computer 640x480, visual control dimensions will be reduced by a factor of 0.625 (640 / 1024 = 0.625). The original font size of 8 is reduced to 5 (8 * 0.625 = 5). Text in the application appears jagged and unreadable as it is displayed in the reduced font size.

- Some visual controls, such as *TLabel* and *TEdit*, dynamically resize when the size of the font for the control changes. This can affect deployed applications when forms and controls are dynamically resized. The resizing of the control due to font size changes are in addition to size changes due to proportional resizing for screen resolutions. This effect is offset by setting the AutoSize property of these controls to *False*.
- Avoid making use of explicit pixel coordinates, such as when drawing directly to a canvas. Instead, modify the coordinates by a ratio proportionate to the screen resolution difference ratio between the development and user computers. For example, if the application draws a rectangle to a canvas ten pixels high by twenty wide, instead multiply the ten and twenty by the screen resolution difference ratio. This ensures that the rectangle visually appears the same size under different screen resolutions.

See Also

Considerations When Not Dynamically Resizing (see page 1947)

Accommodating Varying Color Depths (see page 1945)

3.2.4.4.9 Considerations When Not Dynamically Resizing

If the forms and visual controls that make up an application are not dynamically resized at runtime, design the application's elements for the lowest resolution. Otherwise, the forms of an application run on a computer configured for a lower screen resolution than the development computer may overlap the boundaries of the screen.

For example, if the development computer is set up for a screen resolution of 1024x768 and a form is designed with a width of 700 pixels, not all of that form will be visible within the desktop on a computer configured for a 640x480 screen resolution.

See Also

Considerations When Dynamically Resizing Forms and Controls (see page 1946)

Accommodating Varying Color Depths (see page 1945)

3.2.4.4.10 DLL Locations

You can install DLL files used only by a single application in the same directory as the application. DLLs that will be used by a number of applications should be installed in a location accessible to all of those applications. A common convention for locating such community DLLs is to place them either in the Windows or the Windows\System directory. A better way is to create a dedicated directory for the common .DLL files, similar to the way the Borland Database Engine is installed.

See Also

Using Installation Programs (see page 1954)

Identifying Application Files (see page 1950)

Helper Applications (see page 1950)

3.2.4.4.11 Deploying Applications: Overview

Once your application is up and running, you can deploy it. That is, you can make it available for others to run. A number of steps must be taken to deploy an application to another computer so that the application is completely functional. The steps required by a given application vary, depending on the type of application. The following sections describe these steps when deploying the following applications:

- Deploying General Applications (see page 1949)

- Deploying Database Applications (see page 1948)
- Deploying Web Applications (see page 1949)
- Programming for Varying Host Environments (see page 1953)
- Software License Requirements (see page 1954)

See Also

Overview of Component Creation (see page 1313)

Creating International Applications (see page 1923)

Working with Packages and Components: Overview (see page 2211)

Creating Internet Server Applications (see page 2251)

3.2.4.4.12 Deploying Database Applications

Applications that access databases involve special installation considerations beyond copying the application's executable file onto the host computer. Database access is most often handled by a separate database engine, the files of which cannot be linked into the application's executable file. The data files, when not created beforehand, must be made available to the application. Multi-tier database applications require additional handling on installation, because the files that make up the application are typically located on multiple computers.

Since several different database technologies (ADO, BDE, dbExpress, and InterBase Express) are supported, deployment requirements differ for each. Regardless of which you are using, you need to make sure that the client-side software is installed on the system where you plan to run the database application. ADO, BDE, dbExpress, and InterBase Express also require drivers to interact with the client-side software of the database.

Specific information on how to deploy dbExpress, BDE, and multi-tiered database applications is described in the following topics:

- Deploying dbExpress Database Applications (see page 1942).
- Deploying BDE Applications (see page 1941).
- Deploying Multi-tiered Database Applications (DataSnap) (see page 1952).

Database applications that use client datasets such as *TClientDataSet* or dataset providers require you to include midaslib.dcu (for static linking when providing a stand-alone executable); if you are packaging your application (with the executable and any needed DLLs), you need to include Midas.dll.

If deploying database applications that use ADO, you need to be sure that MDAC version 2.1 or later is installed on the system where you plan to run the application. MDAC is automatically installed with software such as Windows 2000 and Internet Explorer version 5 or later. You also need to be sure the drivers for the database server you want to connect to are installed on the client. No other deployment steps are required.

If deploying database applications that use InterBase Express, you need to be sure that the InterBase client is installed on the system where you plan to run the application. InterBase requires gds32.dll and interbase.msg to be located in an accessible directory. No other deployment steps are required. InterBase Express components communicate directly with the InterBase Client API and do not require additional drivers. For more information, refer to the Embedded Installation Guide posted on the CodeGear Web site.

In addition to the technologies described here, you can also use third-party database engines to provide database access. Consult the documentation or vendor for the database engine regarding redistribution rights, installation, and configuration.

See Also

Deploying General Applications (see page 1949)

Deploying Web Applications (see page 1949)

Programming for Varying Host Environments (see page 1953)

Software License Requirements (see page 1954)

Deploying Packages (see page 2215)

3.2.4.4.13 Deploying General Applications

Beyond the executable file, an application may require a number of supporting files, such as DLLs, package files, and helper applications. In addition, the Windows registry may need to contain entries for an application, from specifying the location of supporting files to simple program settings. The process of copying an application's files to a computer and making any needed registry settings can be automated by an installation program, such as InstallShield Express. Nearly all types of applications include the following issues:

- Using installation programs (see page 1954)
- Identifying application files (see page 1950)
- Helper applications (see page 1950)
- DLL locations (see page 1947)

Database and Web applications require additional installation steps. For additional information on installing database applications, see Deploying database applications (see page 1948). For more information on installing Web applications, see Deploying Web applications (see page 1949).

See Also

Programming for Varying Host Environments (see page 1953)

Software License Requirements (see page 1954)

Deploying Packages (see page 2215)

3.2.4.4.14 Deploying Web Applications

Some Delphi applications are designed to be run over the World Wide Web, such as those in the form of Server-side Extension DLLs (ISAPI and Apache), CGI applications, and ActiveForms.

The steps for deploying Web applications are the same as those for general applications (see page 1949), except the application's files are deployed on the Web server.

Here are some special considerations for deploying Web applications:

- For BDE database applications, the Borland Database Engine (or alternate database engine) is installed with the application files on the Web server.
- For dbExpress applications, the dbExpress DLLs must be included in the path. If included, the *dbExpress* driver is installed with the application files on the Web server.
- Security for the directories should be set so that the application can access all needed database files.
- The directory containing an application must have read and execute attributes.
- The application should not use hard-coded paths for accessing database or other files.
- The location of an ActiveX control is indicated by the CODEBASE parameter of the <OBJECT> HTML tag.

For information on deploying database Web applications, see Deploying database applications (see page 1948).

For information on deploying applications on Apache servers, see Deploying on Apache servers (see page 1943).

See Also

Deploying General Applications (see page 1949)

Deploying Database Applications (see page 1948)

Programming for Varying Host Environments (see page 1953)

Software License Requirements (see page 1954)

Deploying Packages (see page 2215)

Creating Internet Server Applications (see page 2251)

3.2.4.4.15 Fonts

Windows comes with a standard set of TrueType and raster fonts. Linux comes with a standard set of fonts, depending on the distribution. When designing an application to be deployed on other computers, realize that not all computers have fonts outside the standard sets.

Text components used in the application should all use fonts that are likely to be available on all deployment computers.

When use of a nonstandard font is absolutely necessary in an application, you need to distribute that font with the application. Either the installation program or the application itself must install the font on the deployment computer. Distribution of third-party fonts may be subject to limitations imposed by the font creator.

Windows has a safety measure to account for attempts to use a font that does not exist on the computer. It substitutes another, existing font that it considers the closest match. While this may circumvent errors concerning missing fonts, the end result may be a degradation of the visual appearance of the application. It is better to prepare for this eventuality at design time.

To make a nonstandard font available to a Windows application, use the Windows API functions *AddFontResource* and *DeleteFontResource*. Deploy the .fot file for the nonstandard font with the application.

See Also

Screen Resolutions and Color Depths (see page 1953)

Operating System Versions (see page 1955)

3.2.4.4.16 Helper Applications

Helper applications are separate programs without which your application would be partially or completely unable to function. Helper applications may be those supplied with the operating system, by CodeGear, or by third-party products. An example of a helper application is the InterBase utility program Server Manager, which administers InterBase databases, users, and security.

If an application depends on a helper program, be sure to deploy it with your application, where possible. Distribution of helper programs may be governed by redistribution license agreements. Consult the helper program documentation for specific information.

See Also

Using Installation Programs (see page 1954)

Identifying Application Files (see page 1950)

DLL Locations (see page 1947)

3.2.4.4.17 Identifying Application Files

Besides the executable file, a number of other files may need to be distributed with an application.

- Application files (see page 1945)
- Package files (see page 1952)
- Merge modules (see page 1951)
- ActiveX controls (see page 1945)

See Also

Using Installation Programs (see page 1954)

Helper Applications (see page 1950)

DLL Locations (see page 1947)

3.2.4.4.18 Merge Modules

InstallAware Express is based on Windows Installer (MSI) technology. With MSI-based setup tools such as InstallAware Express, you can use merge modules for deploying runtime packages. Merge modules provide a standard method that you can use to deliver shared code, files, resources, Registry entries, and setup logic to applications as a single compound file.

The runtime libraries have some interdependencies because of the way they are grouped together. The result of this is that when one package is added to an install project, the install tool automatically adds or reports a dependency on one or more other packages. For example, if you add the VCLInternet merge module to an install project, the install tool should also automatically add or report a dependency on the VCLDatabase and StandardVCL modules.

The dependencies for each merge module are listed in the table below. The various install tools may react to these dependencies differently. The InstallAware Express for Windows Installer automatically adds the required modules if it can find them. Other tools may simply report a dependency or may generate a build failure if all required modules are not included in the project.

Merge modules and their dependencies

Merge Module	BPLs Included	Dependencies
ADO	adortl<nnn>.bpl	DatabaseRTL, BaseRTL
BaseRTL	rtl<nnn>.bpl	No dependencies
BaseVCL	vcl<nnn>.bpl, vclx<nnn>.bpl	BaseRTL
BDEInternet	inetdbbde<nnn>.bpl	Internet, DatabaseRTL, BaseRTL, BDERTL
BDERTL	bdertl<nnn>.bpl	DatabaseRTL, BaseRTL
DatabaseRTL	dbrtl<nnn>.bpl	BaseRTL
DatabaseVCL	vcldb<nnn>.bpl	BaseVCL, DatabaseRTL, BaseRTL
DataSnap	dsnap<nnn>.bpl	DatabaseRTL, BaseRTL
DataSnapConnection	dsnapcon<nnn>.bpl	DataSnap, DatabaseRTL, BaseRTL
DataSnapEntera	dsnapent<nnn>.bpl	DataSnap, DatabaseRTL, BaseRTL, BaseVCL
DBCompatVCL	vcldbx<nnn>.bpl	DatabaseVCL, BaseVCL, BaseRTL, DatabaseRTL
dbExpress	dbexpress<nnn>.bpl	DatabaseRTL, BaseRTL
dbExpressClientDataSet	dbxcds<nnn>.bpl	BaseClientDataSet, DataBaseRTL, BaseRTL, DataSnap, dbExpress
DBXInternet	inetdbxpress<nnn>.bpl	Internet, DatabaseRTL, BaseRTL, dbExpress, DatabaseVCL, BaseVCL
DecisionCube	dss<nnn>.bpl	TeeChart, BaseVCL, BaseRTL, DatabaseVCL, DatabaseRTL, BDERTL
InterbaseVCL	ibxpress<nnn>.bpl	BaseClientDataSet, BaseRTL, BaseVCL, DatabaseRTL, DatabaseVCL, DataSnap, dbExpress
Internet	inet<nnn>.bpl, inetdb<nnn>.bpl	DatabaseRTL, BaseRTL
InternetDirect	indy<nnn>.bpl	BaseVCL, BaseRTL

Delphi Office2000Components	dcloffice2k<nnn>.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL
C++ Office2000Components	bcboffice2k<nnn>.bpl	BaseVCL, BaseRTL
Delphi OfficeXPComponents	dclofficexp<nnn>.bpl	DatabaseVCL, BaseVCL, DatabaseRTL, BaseRTL
C++ OfficeXPComponents	bcbofficexp<nnn>.bpl	BaseVCL, BaseRTL
SOAPRTL	soaprtl<nnn>.bpl	BaseRTL, XMLRTL, DatabaseRTL, DataSnap, Internet
TeeChart	tee<nnn>.bpl, teedb<nnn>.bpl, teeqr<nnn>.bpl, teeui<nnn>.bpl	BaseVCL, BaseRTL
VCLActionBands	vclactnband<nnn>.bpl	BaseVCL, BaseRTL
VCLIE	vclie<nnn>.bpl	BaseVCL, BaseRTL
WebDataSnap	webdsnap<nnn>.bpl	XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL
WebSnap	websnap<nnn>.bpl, vcljpg<nnn>.bpl	WebDataSnap, XMLRTL, Internet, DataSnapConnection, DataSnap, DatabaseRTL, BaseRTL, BaseVCL
XMLRTL	xmlrtl<nnn>.bpl	Internet, DatabaseRTL, BaseRTL

See Also

Application Files (see page 1945)

Deploying Packages (see page 2215)

Runtime Packages (see page 2220)

Using Installation Programs (see page 1954)

Package Files (see page 1952)

3.2.4.4.19 Deploying Multi-tiered Database Applications (DataSnap)

DataSnap provides multi-tier database capability to Delphi applications by allowing client applications to connect to providers in an application server.

Install DataSnap along with a multi-tier application using InstallAware (or other CodeGear-certified installation scripting utility). See the DEPLOY document (found in the main Delphi directory) for details on the files that need to be redistributed with an application. Also see the REMOTE document for related information on what DataSnap files can be redistributed and how.

See Also

Creating Multi-tiered Applications: Overview (see page 1518)

Using Installation Programs (see page 1954)

3.2.4.4.20 Package Files

If the application uses runtime packages, those package files need to be distributed with the application. InstallShield Express handles the installation of package files the same as DLLs, copying the files and making necessary entries in the Windows registry. You can also use merge modules for deploying runtime packages with MSI-based setup tools including InstallShield Express. See Merge modules (see page 1951) for details.

CodeGear recommends installing the runtime package files supplied by CodeGear in the Windows\System directory. This serves as a common location so that multiple applications would have access to a single instance of the files. For packages you created, it is recommended that you install them in the same directory as the application. Only the .bpl files need to be distributed.

If you are distributing packages to other developers, supply the .bpl and .dcp files.

See Also

Application Files (see page 1945)

Deploying Packages (see page 2215)

Runtime Packages (see page 2220)

Using Installation Programs (see page 1954)

3.2.4.4.21 Programming for Varying Host Environments

Due to the nature of various operating system environments, there are a number of factors that vary with user preference or configuration. The following factors can affect an application deployed to another computer:

- Screen resolutions and color depths (see page 1953)
- Fonts (see page 1950)
- Operating system versions (see page 1955)
- Helper applications (see page 1950)
- DLL locations (see page 1947)

See Also

Deploying General Applications (see page 1949)

Deploying Web Applications (see page 1949)

Deploying Database Applications (see page 1948)

Software License Requirements (see page 1954)

Creating International Applications (see page 1923)

3.2.4.4.22 Screen Resolutions and Color Depths

The size of the desktop and number of available colors on a computer is configurable and dependent on the hardware installed. These attributes are also likely to differ on the deployment computer compared to those on the development computer.

An application's appearance (window, object, and font sizes) on computers configured for different screen resolutions can be handled in various ways:

- Design the application for the lowest resolution users will have (typically, 640x480). Take no special actions to dynamically resize objects to make them proportional to the host computer's screen display. Visually, objects will appear smaller the higher the resolution is set.
- Design using any screen resolution on the development computer and, at runtime, dynamically resize all forms and objects proportional to the difference between the screen resolutions for the development and deployment computers (a screen resolution difference ratio).
- Design using any screen resolution on the development computer and, at runtime, dynamically resize only the application's forms. Depending on the location of visual controls on the forms, this may require the forms be scrollable for the user to be able to access all controls on the forms.

The following topics are discussed in this section:

- Considerations When Not Dynamically Resizing (see page 1947)
- Considerations When Dynamically Resizing Forms and Controls (see page 1946)
- Accommodating Varying Color Depths (see page 1945)

See Also

Fonts (see page 1950)

Operating System Versions (see page 1955)

3.2.4.4.23 Software License Requirements

The distribution of some files associated with Delphi applications is subject to limitations or cannot be redistributed at all. The following documents describe the legal stipulations regarding the distribution of these files:

DEPLOY

The DEPLOY document covers the some of the legal aspects of distributing of various components and utilities, and other product areas that can be part of or associated with a Delphi application. The DEPLOY document is installed in the main Delphi directory. The topics covered include:

- .exe, .dll, and .bpl files
- Components and design-time packages
- Borland Database Engine (BDE)
- ActiveX controls
- Sample images

README

The README document contains last minute information about Delphi, possibly including information that could affect the redistribution rights for components, or utilities, or other product areas. The README document is installed in the main Delphi directory.

No-nonsense license agreement

The Delphi no-nonsense license agreement, a printed document, covers other legal rights and obligations concerning Delphi.

Third-party product documentation

Redistribution rights for third-party components, utilities, helper applications, database engines, and other products are governed by the vendor supplying the product. Consult the documentation for the product or the vendor for information regarding the redistribution of the product with Delphi applications prior to distribution.

See Also

Deploying General Applications (see page 1949)

Deploying Web Applications (see page 1949)

Programming for Varying Host Environments (see page 1953)

Deploying Database Applications (see page 1948)

3.2.4.4.24 Using Installation Programs

Simple applications that consist of only an executable file are easy to install on a target computer. Just copy the executable file onto the computer. However, more complex applications that comprise multiple files require more extensive installation procedures. These applications require dedicated installation programs.

Setup toolkits automate the process of creating installation programs, often without needing to write any code. Installation programs created with Setup toolkits perform various tasks inherent to installing Delphi applications, including: copying the executable and supporting files to the host computer, making Windows registry entries, and installing the Borland Database Engine for BDE database applications.

InstallAware Express is a setup toolkit that is bundled with Delphi. InstallAware Express is certified for use with Delphi and the Borland Database Engine. It is based on Windows Installer (MSI) technology.

InstallAware Express is not automatically installed when Delphi is installed, so it must be manually installed if you want to use it to create installation programs. Run the installation program from the Delphi CD to install InstallAware Express. For more information on using InstallAware Express to create installation programs, see the InstallAware Express online help.

Other setup toolkits are available. However, if deploying BDE database applications, you should only use toolkits based on MSI technology and those which are certified to deploy the Borland Database Engine.

See Also

Identifying Application Files (see page 1950)

Helper Applications (see page 1950)

DLL Locations (see page 1947)

3.2.4.4.25 Operating System Versions

When using operating system APIs or accessing areas of the operating system from an application, there is the possibility that the function, operation, or area may not be available on computers with different operating system versions.

To account for this possibility, you have a few options:

- Specify in the application's system requirements the versions of the operating system on which the application can run. It is the user's responsibility to install and use the application only under compatible operating system versions.
- Check the version of the operating system as the application is installed. If an incompatible version of the operating system is present, either halt the installation process or at least warn the installer of the problem.
- Check the operating system version at runtime, just prior to executing an operation not applicable to all versions. If an incompatible version of the operating system is present, abort the process and alert the user. Alternately, provide different code to run dependent on different operating system versions.

  Note: Some operations are performed differently on Windows 95/98 than on Windows NT/2000/XP. Use the Windows API function *GetVersionEx* to determine the Windows version.

See Also

Screen Resolutions and Color Depths (see page 1953)

Fonts (see page 1950)

3.2.4.5 Developing the application user interface

Topics

Name	Description
Adding Forms (see page 1965)	To add a form to your project, select **File** ▶ **New** ▶ **Form**. You can see all your project's forms and their associated units listed in the **Project Manager** (**View** ▶ **Project Manager**) and you can display a list of the forms alone by choosing **View** ▶ **Forms**.

Adding Color, Patterns, or Pictures to Menus, Buttons, and Toolbars (see page 1966)	You can use the *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on a menu item or button. These properties also let you set up a banner the runs up the left or right side of a menu. You assign backgrounds and layouts to subitems from their action client objects. If you want to set the background of the items in a menu, in the form designer click on the menu item that contains the items. For example, selecting File lets you change the background of items appearing on the File menu. You can assign... more (see page 1966)
Adding Icons to Menus and Toolbars (see page 1966)	You can add icons next to menu items or replace captions on toolbars with icons. You organize bitmaps or icons using an *ImageList* component.
Adding Images to Menu Items (see page 1967)	Images can help users navigate in menus by matching glyphs and images to menu item action, similar to toolbar images. You can add single bitmaps to menu items, or you can organize images for your application into an image list and add them to a menu from the image list. If you're using several bitmaps of the same size in your application, it's useful to put them into an image list. To add a single image to a menu or menu item, set its *Bitmap* property to reference the name of the bitmap to use on the menu or menu... more (see page 1967)
Adding, Inserting, and Deleting Menu Items (see page 1968)	The following procedures describe how to perform the basic tasks involved in building your menu structure. Each procedure assumes you have the Menu Designer window open.
Adding a Cool Bar Component (see page 1968)	**Note:** The *TCoolBar* component requires version 4.70 or later of COMCTL32.DLL. The cool bar component (*TCoolBar*)—also called a *rebar*—displays windowed controls on independently movable, resizable bands. The user can position the bands by dragging the resizing grips on the left side of each band.
Adding a Speed Button to a Panel (see page 1969)	To add a speed button to a toolbar panel, place the speed button component (from the Additional category of the **Tool palette**) on the panel. The panel, rather than the form, "owns" the speed button, so moving or hiding the panel also moves or hides the speed button. The default height of the panel is 41, and the default height of speed buttons is 25. If you set the *Top* property of each button to 8, they'll be vertically centered. The default grid setting snaps the speed button to that vertical position for you.
Adding a Toolbar Using a Panel Component (see page 1969)	
Adding a Toolbar Using the Toolbar Component (see page 1970)	The toolbar component (*TToolBar*) offers button management and display features that panel components do not.
Adding a Tool Button (see page 1971)	To add a tool button to a toolbar, right-click on the toolbar and choose New Button. The toolbar "owns" the tool button, so moving or hiding the toolbar also moves or hides the button. In addition, all tool buttons on the toolbar automatically maintain the same height and width. You can drop other controls from the **Tool palette** onto the toolbar, and they will automatically maintain a uniform height. Controls will also wrap around and start a new row when they do not fit horizontally on the toolbar.
Adding Hidden Toolbars (see page 1971)	Toolbars do not have to be visible all the time. In fact, it is often convenient to have a number of toolbars available, but show them only when the user wants to use them. Often you create a form that has several toolbars, but hide some or all of them.
Allowing Toggle Buttons (see page 1971)	Sometimes you want to be able to click a button in a group that's already pressed and have it pop up, leaving no button in the group pressed. Such a button is called a toggle. Use *AllowAllUp* to create a grouped button that acts as a toggle: click it once, it's down; click it again, it pops up. To make a grouped speed button a toggle, set its *AllowAllUp* property to *True*. Setting *AllowAllUp* to *True* for any speed button in a group automatically sets the same property value for all buttons in the group. This enables the group... more (see page 1971)
Allowing Toggled Tool Buttons (see page 1972)	Use *AllowAllUp* to create a grouped tool button that acts as a toggle: click it once, it is down; click it again, it pops up. To make a grouped tool button a toggle, set its *AllowAllUp* property to *True*. As with speed buttons, setting *AllowAllUp* to *True* for any tool button in a group automatically sets the same property value for all buttons in the group.
Assigning a Menu to a Tool Button (see page 1972)	If you are using a toolbar (*TToolBar*) with tool buttons (*TToolButton*), you can associate menu with a specific button:

Assigning a Speed Button's Glyph (see page 1972)	Each speed button needs a graphic image called a *glyph* to indicate to the user what the button does. If you supply the speed button only one image, the button manipulates that image to indicate whether the button is pressed, unpressed, selected, or disabled. You can also supply separate, specific images for each state if you prefer. You normally assign glyphs to speed buttons at design time, although you can assign different glyphs at runtime.
Assigning Images to Tool Buttons (see page 1973)	Each tool button has an *ImageIndex* property that determines what image appears on it at runtime. If you supply the tool button only one image, the button manipulates that image to indicate whether the button is disabled.
Building Menus (see page 1973)	You add a menu component to your form, or forms, for every menu you want to include in your application. You can build each menu structure entirely from scratch, or you can start from one of the predesigned menu templates. For more information about menu templates, see Using menu templates (see page 2009). For more information about creating a menu using the menu designer see • Naming menus (see page 1991) • Naming the menu items (see page 1991) • Adding (see page 1968) • Creating submenus (see page 1978) • Adding images to menu items (see page 1967) • Viewing the menu (see page 2011)
Common Controls and XP Themes (see page 1974)	Microsoft has forked Windows common controls into two separate versions. Version 5 is available on all Windows versions from Windows 95 or later; it displays controls using a "3D chiseled" look. Version 6 became available with Windows XP. Under version 6, controls are rendered by a theme engine which matches the current Windows XP theme. If the user changes the theme, version 6 common controls will match the new theme automatically. You don't need to recompile the application. The VCL can now accommodate both types of common controls. CodeGear has added a number of components to the VCL to handle... more (see page 1974)
Controlling When Forms Reside in Memory (see page 1975)	By default, Delphi automatically creates the application's main form in memory by including the following code in the application's main entry point:
Creating and Managing Menus (see page 1975)	Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predesigned or custom tailored—to your form. You add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. You can add or delete menu items, or drag and drop them to rearrange them during design time. You don't even need to run your program to see the results—your design is immediately visible in the form, appearing just as it will during runtime. Your code can also... more (see page 1975)
Creating and Using Component Templates (see page 1975)	You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, save them as a *component template*. Later, by selecting the template from the **Tool palette**, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time. Once you place a template on a form, you can reposition the components independently, reset their properties, and create or modify event handlers for them just as if... more (see page 1975)
Creating Customizable Toolbars and Menus (see page 1976)	You can use action bands with the Action Manager to create customizable toolbars and menus. At runtime, users of your application can customize the toolbars and menus (action bands) in the application user interface using a customization dialog similar to the Action Manager editor.
Creating Dynamic Menus (see page 1977)	Dynamic menus and toolbars allow users to modify the application in various ways at run time. Some examples of dynamic usage include customizing the appearance of toolbars and menus, hiding unused items, and responding to most recently used lists (MRUs).

Creating Frames (see page 1977)	To create an empty frame, choose **File** ▶ **New** ▶ **Delphi Projects** ▶ **Delphi Files** and double-click Frame. You can then drop components (including other frames) onto your new frame. It is usually best—though not necessary—to save frames as part of a project. If you want to create a project that contains only frames and no forms, choose **File** ▶ **New** ▶ **Application**, close the new form and unit without saving them, then choose **File** ▶ **New** ▶ **Delphi Projects** ▶ **Delphi Files** ▶ **Frame** and save the project. **Note:** When you save frames, avoid using the default names *Unit1*, *Project1*, and so forth, since these... more (see page 1977)
Creating Most Recently Used Lists (see page 1977)	A most recently used list (MRU) reflects the user's most recently accessed files in a specific application. Using action bands, you can code MRU lists in your applications. When building MRUs for your applications, it is important not to hard code references to specific numerical indexes into the Action Manager's *ActionBars* property. At runtime, the user may change the order of items or even delete them from the action bands, which in turn will change the numerical ordering of the index. Instead of referring to index numbering, *TActionManager* includes methods that facilitate finding items by action or by caption. For... more (see page 1977)
Creating Submenus (see page 1978)	Many application menus contain drop-down lists that appear next to a menu item to provide additional, related commands. Such lists are indicated by an arrow to the right of the menu item. Delphi supports as many levels of such submenus as you want to build into your menu. Organizing your menu structure this way can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one submenu, if any.)
Creating Toolbars and Menus (see page 1978)	**Note:** This topic describes the recommended method for creating menus and toolbars in Windows applications. For cross-platform development, you need to use *TToolBar* and the menu components, such as *TMainMenu,* organizing them using action lists (*TActionList*). See Setting up action lists (see page 2003) for details. You use the Action Manager to automatically generate toolbars and main menus based on the actions contained in your application. The Action Manager manages standard actions and any custom actions that you have written. You then create UI elements based on these actions and use action bands to render the actions items as either menu... more (see page 1978)
Creating a Group of Speed Buttons (see page 1979)	A series of speed buttons often represents a set of mutually exclusive choices. In that case, you need to associate the buttons into a group, so that clicking any button in the group causes the others in the group to pop up. To associate any number of speed buttons into a group, assign the same number to each speed button's *GroupIndex* property. The easiest way to do this is to select all the buttons you want in the group, and, with the whole group selected, set *GroupIndex* to a unique value.
Creating Forms Dynamically (see page 1980)	You may not always want all your application's forms in memory at once. To reduce the amount of memory required at load time, you may want to create some forms only when you need to use them. For example, a dialog box needs to be in memory only during the time a user interacts with it.
Creating Groups of Tool Buttons (see page 1981)	To create a group of tool buttons, select the buttons you want to associate and set their *Style* property to *tbsCheck*; then set their *Grouped* property to *True*. Selecting a grouped tool button causes other buttons in the group to pop up, which is helpful to represent a set of mutually exclusive choices. Any unbroken sequence of adjacent tool buttons with *Style* set to *tbsCheck* and *Grouped* set to *True* forms a single group. To break up a group of tool buttons, separate the buttons with any of the following: • A tool button whose *Grouped* property is *False... more (see page 1981)*
Creating Modeless Forms Such as Windows (see page 1981)	You must guarantee that reference variables for modeless forms exist for as long as the form is in use. This means that these variables should have global scope. In most cases, you use the global reference variable that was created when you made the form (the variable name that matches the name property of the form). If your application requires additional instances of the form, declare separate global variables for each instance.

Demo Programs: Actions, Action Lists, Menus, and Toolbars (see page 1981)	For examples of Windows applications that use actions, action lists, menus, and toolbars, refer to <InstallDrive>\Documents and Settings\All Users\Documents\RAD Studio\6.0\Demos\DelphiWin32\VCLWin32\RichEdit (Windows XP), <InstallDrive>\Users\All Users\Documents\RAD Studio\6.0\Demos\DelphiWin32\VCLWin32\RichEdit (Windows Vista). In addition, the Application wizard (**File** ▶ **New** ▶ **Other**), MDI Application, SDI Application, and Winx Logo Applications can use the action and action list objects.
Designing Toolbars and Cool Bars (see page 1982)	A *toolbar* is a panel, usually across the top of a form (under the menu bar), that holds buttons and other controls. A *cool bar* (also called a rebar) is a kind of toolbar that displays controls on movable, resizable bands. If you have multiple panels aligned to the top of the form, they stack vertically in the order added. You can put controls of any sort on a toolbar. In addition to buttons, you may want to put use color grids, scroll bars, labels, and so on. You can add a toolbar to a form in several ways: • Place... more (see page 1982)
Determining the Order of Merged Menu Items: GroupIndex Property (see page 1982)	The *GroupIndex* property determines the order in which the merging menu items appear in the shared menu bar. Merging menu items can replace those on the main menu bar, or can be inserted. The default value for *GroupIndex* is 0. Several rules apply when specifying a value for *GroupIndex*:
Developing the Application User Interface: Overview (see page 1983)	When you open the IDE or create a new project, a blank form is displayed on the screen. You design your application's user interface (UI) by placing and arranging visual components, such as windows, menus, and dialog boxes, from the **Tool palette** onto the form. Once a visual component is on the form, you can adjust its position, size, and other design-time properties, and code its event handlers (see page 2149). The form takes care of the underlying programming details. The following topics describe some of the major interface tasks, such as working with forms, creating component templates, adding dialog boxes, and... more (see page 1983)
Displaying an Auto-created Form (see page 1983)	If you choose to create a form at startup, and do not want it displayed until sometime later during program execution, the form's event handler uses the *ShowModal* method to display the form that is already loaded in memory:
Editing Menu Items in the Object Inspector (see page 1983)	This topic has discussed how to set several properties for menu items—for example, the *Name* and *Caption* properties—by using the Menu Designer. The section has also described how to set menu item properties, such as the *ShortCut* property, directly in the **Object Inspector**, just as you would for any component selected in the form. When you edit a menu item by using the Menu Designer, its properties are still displayed in the **Object Inspector**. You can switch focus to the **Object Inspector** and continue editing the menu item properties there. Or you can select the menu item from... more (see page 1983)
What Happens When an Action Fires (see page 1984)	When an event fires, a series of events intended primarily for generic actions occurs. Then if the event doesn't handle the action, another sequence of events occurs.
Handling the Screen (see page 1985)	A global variable of type *TScreen* called *Screen* is created when you create a project. Screen encapsulates the state of the screen on which your application is running. Common tasks performed by *Screen* include specifying: • The look of the cursor. • The size of the window in which your application is running. • A list of fonts available to the screen device. • Multiple screen behavior (Windows only). If your Windows application runs on multiple monitors, *Screen* maintains a list of monitors and their dimensions so that you can effectively manage the layout of your user interface.
Hiding the Main Form (see page 1986)	You can prevent the main form from appearing when your application starts by using the global Application variable (see page 2012).

Hiding Unused Items and Categories in Action Bands (see page 1986)	One benefit of using ActionBands is that unused items and categories can be hidden from the user. Over time, the action bands become customized for the application users, showing only the items that they use and hiding the rest from view. Hidden items can become visible again when the user presses a drop-down button. Also, the user can restore the visibility of all action band items by resetting the usage statistics from the customization dialog. Item hiding is the default behavior of action bands, but that behavior can be changed to prevent hiding of individual items, all the items in... more (see page 1986)
Hiding and Showing Toolbars (see page 1987)	Often, you want an application to have multiple toolbars, but you do not want to clutter the form with them all at once. Or you may want to let users decide whether to display toolbars. As with all components, toolbars can be shown or hidden at runtime as needed. To show or hide a toolbar at runtime, set its *Visible* property to *False* or *True*, respectively. Usually you do this in response to particular user events or changes in the operating mode of the application. To do this, you typically have a close button on each toolbar. When the... more (see page 1987)
How Actions Find Their Targets (see page 1987)	What happens when an action fires (see page 1984) describes the execution cycle that occurs when a user invokes an action. If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself. The application looks for the target using the following sequence: 1. Active control: The application looks first for an active control as a potential target. 2. Active form: If the application does not find an active control or if the active control can't act as a target, it... more (see page 1987)
Developing Dialog Boxes (see page 1988)	The dialog box components on the Dialogs category of the **Tool palette** make various dialog boxes available to your applications. These dialog boxes provide applications with a familiar, consistent interface that enables the user to perform common file operations such as opening, saving, and printing files. Dialog boxes display and/or obtain data. Each dialog box opens when its *Execute* method is called. *Execute* returns a Boolean value: if the user chooses OK to accept any changes made in the dialog box, *Execute* returns *True*; if the user chooses Cancel to escape from the dialog box without making or... more (see page 1988)
Importing Resource Files (see page 1988)	You can build menus with other applications, so long as the menus are in the standard Windows resource (.RC) file format. You can import such menus directly into your project, saving you the time and effort of rebuilding menus that you created elsewhere.
Managing Layout (see page 1988)	At its simplest, you control the layout of your user interface by where you place controls in your forms. The placement choices you make are reflected in the control's *Top*, *Left*, *Width*, and *Height* properties. You can change these values at runtime to change the position and size of the controls in your forms. Controls have a number of other properties, however, that allow them to automatically adjust to their contents or containers. This allows you to lay out your forms so that the pieces fit together into a unified whole. Two properties affect how a... more (see page 1988)
Manipulating Menu Items at Runtime (see page 1989)	Sometimes you want to add menu items to an existing menu structure while the application is running, to provide more information or options to the user. You can insert a menu item by using the menu item's *Add* or *Insert* method, or you can alternately hide and show the items in a menu by changing their *Visible* property. The *Visible* property determines whether the menu item is displayed in the menu. To dim a menu item without hiding it, use the Enabled property. For examples that use the menu item's *Visible* and *Enabled* properties, see Disabling menu items (see page 2160). In... more (see page 1989)
Merging Menus (see page 1989)	For MDI applications, such as the text editor sample application, and for OLE client applications, your application's main menu needs to be able to receive menu items either from another form or from the OLE server object. This is often called *merging menus*. Note that OLE technology is limited to Windows applications only and is not available for use in cross-platform programming. You prepare menus for merging by specifying values for two properties: • *Menu*, a property of the form • *GroupIndex*, a property of menu items in the menu

Using Forms (see page 1990)	When you create a form from the IDE, Delphi automatically creates the form in memory by including code in the main entry point of your application function. Usually, this is the desired behavior and you don't have to do anything to change it. That is, the main window persists through the duration of your program, so you would likely not change the default behavior when creating the form for your main window. However, you may not want all your application's forms in memory for the duration of the program execution. That is, if you do not want all your application's... more (see page 1990)
Moving Menu Items (see page 1990)	During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or into a *different* menu entirely. The only exception to this is hierarchical: you cannot demote a menu item from the menu bar into its own menu; nor can you move a menu item into its own submenu. However, you can move any item into a different menu, no matter what its original position is. While you are dragging, the cursor changes shape to indicate whether you can release... more (see page 1990)
Naming Conventions for Template Menu Items and Event Handlers (see page 1990)	When you save a menu as a template, Delphi does not save its *Name* property, since every menu must have a unique name within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, Delphi then generates new names for it and all of its items. For example, suppose you save a File menu as a template. In the original menu, you name it *MyFile*. If you insert it as a template into a new menu, Delphi names it *File1*. If you insert... more (see page 1990)
Naming Menus (see page 1991)	As with all components, when you add a menu component to the form, the form gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows language naming conventions. The menu name is added to the form's type declaration, and the menu name then appears in the Component list.
Naming the Menu Items (see page 1991)	In contrast to the menu component itself, you need to explicitly name menu items as you add them to the form. You can do this in one of two ways: • Directly type the value for the *Name* property. • Type the value for the *Caption* property first, and let Delphi derive the *Name* property from the caption. For example, if you give a menu item a *Caption* property value of File, Delphi assigns the menu item a *Name* property of *File1*. If you fill in the *Name* property before filling in the *Caption* property, Delphi leaves the *Caption* property blank... more (see page 1991)
Opening the Menu Designer (see page 1992)	You design menus for your application using the Menu Designer. Before you can start using the Menu Designer, first add either a *TMainMenu* or *TPopupMenu* component to your form. Both menu components are located on the Standard category of the **Tool palette**.  A MainMenu component creates a menu that's attached to the form's title bar. A PopupMenu component creates a menu that appears when the user right-clicks in the form. Pop-up menus do not have a menu bar. To open the Menu Designer, select a menu component on the form, and then either: • Double-click the... more (see page 1992)
Organizing Actions for Toolbars and Menus (see page 1992)	Several features simplify the work of creating, customizing, and maintaining menus and toolbars. These features allow you to organize lists of actions that users of your application can initiate by pressing a button on a toolbar, choosing a command on a menu, or pointing and clicking on an icon. Often a set of actions is used in more than one user interface element. For example, the Cut, Copy, and Paste commands often appear on both an Edit menu and on a toolbar. You only need to add the action once to use it in multiple UI elements in your application.... more (see page 1992)

Passing Additional Arguments to Forms (see page 1993)	Typically, you create forms for your application from within the IDE. When created this way, the forms have a constructor that takes one argument, *Owner*, which is the owner of the form being created. (The owner is the calling application object or form object.) *Owner* can be **nil**. To pass additional arguments to a form, create a separate constructor and instantiate the form using this new constructor. The example form class below shows an additional constructor, with the extra argument *whichButton*. This new constructor is added to the form class manually.
Predefined Action Classes (see page 1995)	You can add predefined actions to your application by right-clicking on the Action Manager and choosing New Standard Action. The New Standard Action Classes dialog box is displayed listing the predefined action classes and the associated standard actions. These are actions that are included with Delphi and they are objects that automatically perform actions. The predefined actions are organized within the following classes:
Registering Actions (see page 1996)	When you write your own actions, you can register actions to enable them to appear in the Action List editor. You register and unregister actions by using the global routines in the Actnlist unit:
Responding to Clicks (see page 1996)	When the user clicks a control, such as a button on a toolbar, the application generates an *OnClick* event which you can respond to with an event handler. Since *OnClick* is the default event for buttons, you can generate a skeleton handler for the event by double-clicking the button at design time. For general information about events and event handlers, see Working with Events and Event Handlers (see page 2149) and Generating a handler for a component's default event (see page 2147).
Retrieving Data from Forms (see page 1997)	Most real-world applications consist of several forms. Often, information needs to be passed between these forms. Information can be passed to a form by means of parameters to the receiving form's constructor, or by assigning values to the form's properties. The way you get information from a form depends on whether the form is modal or modeless.
Retrieving Data from Modal Forms (see page 1997)	Just like modeless forms, modal forms often contain information needed by other forms. The most common example is when form A launches modal form B. When form B is closed, form A needs to know what the user did with form B to decide how to proceed with the processing of form A. If form B is still in memory, it can be queried through properties or member functions just as in the modeless forms example above. But how do you handle situations where form B is deleted from memory upon closing? Since a form does not have an explicit... more (see page 1997)
Retrieving Data from Modeless Forms (see page 1999)	You can easily extract information from modeless forms by calling public member functions of the form or by querying properties of the form. For example, assume an application contains a modeless form called *ColorForm* that contains a listbox called *ColorListBox* with a list of colors ("Red," "Green," "Blue," and so on). The selected color name string in *ColorListBox* is automatically stored in a property called *CurrentColor* each time a user selects a new color. The class declaration for the form is as follows:
Reusing Components and Groups of Components (see page 2000)	You can save and reuse work you've done with components using several tools: • Configure and save groups of components in component templates (see page 1975). • Save forms, data modules, and projects in the Object Repository (see page 1917). The Repository gives you a central database of reusable elements and lets you use form inheritance to propagate changes. • Save frames (see page 2012) on the **Tool palette** or in the Repository. Frames use form inheritance and can be embedded into forms or other frames. • Create a custom component (see page 1313), the most complicated but most flexible way of reusing code. See Overview of Component Creation (see page 1313).
Saving a Menu as a Template (see page 2001)	Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize. The menu templates you save are stored in your BIN subdirectory as .dmt files.

Selecting Menu and Toolbar Styles (see page 2001)	Just as you can add different colors and icons to individual menus and toolbars, you can select different menu and toolbar styles to give your application a comprehensive look and feel. In addition to the standard style, your application can take on the look of Windows XP, Encarta™, or a custom presentation using a coordinated color scheme. To give your application a coherent look and feel, the IDE uses colormaps. A colormap can be simple, merely adding the appropriate colors to existing menus and toolbars. Or, a colormap can be complex, altering numerous subtle details of a menu's or toolbar's... more (see page 2001)
Setting Up Action Bands (see page 2002)	Because actions do not maintain any "layout" (either appearance or positional) information, Delphi provides action bands which are capable of storing this data. Action bands provide a mechanism that allows you to specify layout information and a set of controls. You can render actions as UI elements such as toolbars and menus. You organize sets of actions using the Action Manager (TActionManager). You can use standard actions provided or create new actions of your own. You then create the action bands: • Use TActionMainMenuBar to create a main menu. • Use TActionToolBarto create a toolbar. The action bands act as containers... more (see page 2002)
Setting Up Action Lists (see page 2003)	Setting up action lists is fairly easy once you understand the basic steps involved: • Create the action list. • Add actions to the action list. • Set properties on the actions. • Attach clients to the action.
Setting the Appearance of the Cool Bar (see page 2003)	The cool bar component offers several useful configuration options. The table below lists some actions you can set to change a tool button's appearance: ***Setting a cool button's appearance***
Setting the Initial Condition of a Speed Button (see page 2004)	Speed buttons use their appearance to give the user clues as to their state and purpose. Because they have no caption, it's important that you use the right visual cues to assist users. The table below lists some actions you can set to change a speed button's appearance: ***Setting speed buttons' appearance***
Setting Tool Button Appearance and Initial Conditions (see page 2004)	The table below lists some actions you can set to change a tool button's appearance: ***Setting tool buttons' appearance***
Sharing Frames (see page 2005)	You can share a frame with other developers in two ways: • Add the frame to the Object Repository. • Distribute the frame's unit (.pas) and form (.dfm or .xfm) files. To add a frame to the Repository, open any project that includes the frame, right-click in the Form Designer, and choose Add to Repository. For more information, see Using the Object Repository (see page 1917). If you send a frame's unit and form files to other developers, they can open them and add them to the **Tool palette**. If the frame has other frames embedded in it, they will have to open... more (see page 2005)

Specifying Accelerator Keys and Keyboard Shortcuts (see page 2005)	Accelerator keys enable the user to access a menu command from the keyboard by pressing Alt+ the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu. Delphi automatically checks for duplicate accelerators and adjusts them at runtime. This ensures that menus built dynamically at runtime contain no duplicate accelerators and that all menu items have an accelerator. You can turn off this automatic checking by setting the *AutoHotkeys* property of a menu item to *maManual*. To specify an accelerator, add an ampersand in front of the appropriate... more (see page 2005)
Specifying the Active Menu: Menu Property (see page 2006)	The *Menu* property specifies the active menu for the form. Menu-merging operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at runtime by setting the *Menu* property in code. For example,
Switching Between Menus at Design Time (see page 2006)	If you're designing several menus for your form, you can use the Menu Designer context menu or the **Object Inspector** to easily select and move among them.
Controlling Application Behavior (see page 2007)	TApplication, TScreen, and TForm are the classes that form the backbone of all applications by controlling the behavior of your project. The *TApplication* class forms the foundation of an application by providing properties and methods that encapsulate the behavior of a standard program. *TScreen* is used at runtime to keep track of forms and data modules that have been loaded as well as maintaining system-specific information such as screen resolution and available display fonts. Instances of the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes in your application are based on *TForm... more (see page 2007)*
Updating Actions (see page 2007)	When the application is idle, the *OnUpdate* event occurs for every action that is linked to a control or menu item that is showing. This provides an opportunity for applications to execute centralized code for enabling and disabling, checking and unchecking, and so on. For example, the following code illustrates the *OnUpdate* event handler for an action that is "checked" when the toolbar is visible:
Using Action Lists (see page 2007)	**Note:** The contents of this topic apply to setting up toolbars and menus for cross-platform development. For Windows development you can also use the methods described here. However, using action bands instead is simpler and offers more options. The action lists will be handled automatically by the Action Manager. See Organizing actions for toolbars and menus (see page 1992) for details. Action lists maintain a list of actions that your application can take in response to something a user does. By using action objects, you centralize the functions performed by your application from the user interface. This lets you share common code for... more (see page 2007)
Using and Modifying Frames (see page 2008)	To use a frame in an application, you must place it, directly or indirectly, on a form. You can add frames directly to forms, to other frames, or to other container objects such as panels and scroll boxes. The Form Designer provides two ways to add a frame to an application: • Select a frame from the **Tool palette** and drop it onto a form, another frame, or another container object. If necessary, the Form Designer asks for permission to include the frame's unit file in your project. • Select *Frames* from the Standard category of the **Tool palette** and click on... more (see page 2008)
Using Menu Templates (see page 2009)	Several predesigned menus, or menu templates, contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code. The menu templates are stored in the BIN subdirectory in a default installation and have a .dmt extension. You can also save as a template any menu that you design using the Menu Designer. After saving a menu as a template, you can use it as... more (see page 2009)
Using the Main Form (see page 2009)	The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form. Also, specifying a form as the main form is an easy way to test it at runtime, because unless you change the form creation order, the main form is the first form displayed in the running application.

Using the Menu Designer Context Menu (see page 2010)	The Menu Designer context menu provides quick access to the most common Menu Designer commands, and to the menu template options. (For more information about menu templates, refer to Using menu templates (see page 2009).) To display the context menu, right-click the Menu Designer window, or press *Alt+F10* when the cursor is in the Menu Designer window.
Creating a Form Instance Using a Local Variable (see page 2010)	A safer way to create a unique instance of a *modal form* is to use a local variable in the event handler as a reference to a new instance. If a local variable is used, it does not matter whether *ResultsForm* is auto-created or not. The code in the event handler makes no reference to the global form variable. For example:
Viewing the Menu (see page 2011)	You can view your menu in the form at design time without first running your program code. (Pop-up menu components are visible in the form at design time, but the pop-up menus themselves are not. Use the Menu Designer to view a pop-up menu at design time.)
What Is an Action? (see page 2011)	As you are developing your application, you can create a set of actions that you can use on various UI elements. You can organize them into categories that can be dropped onto a menu as a set (for example, Cut, Copy, and Paste) or one at a time (for example, **Tools** ▶ **Customize**). An action corresponds to one or more elements of the user interface, such as menu commands or toolbar buttons. Actions serve two functions: (1) they represent properties common to the user interface elements, such as whether a control is enabled or checked, and (2) they respond... more (see page 2011)
Working at the Application Level (see page 2012)	The global variable *Application*, of type *TApplication*, is in every VCL-based application. *Application* encapsulates your application as well as providing many functions that occur in the background of the program. For instance, *Application* handles how you call a Help file from the menu of your program. Understanding how *TApplication* works is more important to a component writer than to developers of stand-alone applications, but you should set the options that *Application* handles in the **Project** ▶ **Options** Application page when you create a project. In addition, *Application* receives many events that apply to the application as a whole. For... more (see page 2012)
Working with Frames (see page 2012)	A frame (*TFrame*), like a form, is a container for other components. It uses the same ownership mechanism as forms for automatic instantiation and destruction of the components on it, and the same parent-child relationships for synchronization of component properties. In some ways, however, a frame is more like a customized component than a form. Frames can be saved on the **Tool palette** for easy reuse, and they can be nested within forms, other frames, or other container objects. After a frame is created and saved, it continues to function as a unit and to inherit changes from... more (see page 2012)
Writing Action Components (see page 2013)	You can also create your own predefined action classes. When you write your own action classes, you can build in the ability to execute on certain target classes of objects. Then, you can use your custom actions in the same way you use predefined action classes. That is, when the action can recognize and apply itself to a target class, you can simply assign the action to a client control, and it acts on the target with no need to write an event handler. Component writers can use the classes in the QStdActns and DBActns units as examples for... more (see page 2013)
Using Windows Common Dialog Boxes (see page 2013)	One of the commonly used dialog box components is *TOpenDialog*. This component is usually invoked by a New or Open menu item under the File option on the main menu bar of a form. The dialog box contains controls that let you select groups of files using a wildcard character and navigate through directories. The *TOpenDialog* component makes an Open dialog box available to your application. The purpose of this dialog box is to let a user specify a file to open. You use the *Execute* method to display the dialog box. When the user chooses OK in the... more (see page 2013)

3.2.4.5.1 Adding Forms

To add a form to your project, select **File** ▶ **New** ▶ **Form**. You can see all your project's forms and their associated units listed in the **Project Manager** (**View** ▶ **Project Manager**) and you can display a list of the forms alone by choosing **View** ▶ **Forms**.

Linking forms

Adding a form to a project adds a reference to it in the project file, but not to any other units in the project. Before you can write code that references the new form, you need to add a reference to it in the referencing forms' unit files. This is called *form linking*.

A common reason to link forms is to provide access to the components in that form. For example, you'll often use form linking to

enable a form that contains data-aware components to connect to the data-access components in a data module.

To link a form to another form:

1. Select the form that needs to refer to another.
2. Choose **File ▸ Use Unit**.
3. Select the name of the form unit for the form to be referenced.
4. Choose OK.

Linking a form to another just means that the **uses** clauses of one form unit contains a reference to the other's form unit, meaning that the linked form and its components are now in scope for the linking form.

Avoiding circular unit references

When two forms must reference each other, it's possible to cause a "Circular reference" error when you compile your program. To avoid such an error, do one of the following:

- Place both **uses** clauses, with the unit identifiers, in the **implementation** parts of the respective unit files. (This is what the **File ▸ Use Unit** command does.)
- Place one **uses** clause in an **interface** part and the other in an **implementation** part. (You rarely need to place another form's unit identifier in this unit's **interface** part.)

Do not place both **uses** clauses in the **interface** parts of their respective unit files. This generates the "Circular reference" error at compile time.

See Also

TForm

3.2.4.5.2 Adding Color, Patterns, or Pictures to Menus, Buttons, and Toolbars

You can use the *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on a menu item or button. These properties also let you set up a banner the runs up the left or right side of a menu.

You assign backgrounds and layouts to subitems from their action client objects. If you want to set the background of the items in a menu, in the form designer click on the menu item that contains the items. For example, selecting File lets you change the background of items appearing on the File menu. You can assign a color, pattern, or bitmap in the *Background* property in the **Object Inspector**.

Use the *BackgroundLayout* property to describe how to place the background on the element. Colors or images can be placed behind the caption normally, stretched to fit the item area, or tiled in small squares to cover the area.

Items with normal (blNormal), stretched (blStretch), or tiled (blTile) backgrounds are rendered with a transparent background. If you create a banner, the full image is placed on the left (blLeftBanner) or the right (blRightBanner) of the item. You need to make sure it is the correct size because it is not stretched or shrunk to fit.

To change the background of an action band (that is, on a main menu or toolbar), select the action band and choose the *TActionClientBar* through the action band collection editor. You can set *Background* and *BackgroundLayout* properties to specify a color, pattern, or bitmap to use on the entire toolbar or menu.

See Also

Setting Up Action Lists (see page 2003)

3.2.4.5.3 Adding Icons to Menus and Toolbars

You can add icons next to menu items or replace captions on toolbars with icons. You organize bitmaps or icons using an *ImageList* component.

To add icons to menus and toolbars:

1. Drop an *ImageList* component from the Win32 category of the **Tool palette** onto a form.
2. Add the images you want to use to the image list: Double-click the *ImageList* icon. Click Add and navigate to the images you want to use and click OK when done. Some sample images are included in Program Files\Common Files\Borland Shared\Images. (The buttons images include two views of each for active and inactive buttons.)
3. From the Additional category of the **Tool palette**, drop one or more of the following action bands onto the form:

- TActionMainMenuBar(for designing main menus)
- TActionToolBar *(for designing* toolbars)

4. Connect the image list to the Action Manager. First, set the focus on the Action Manager. Next, in the **Object Inspector**, select the name of the image list from the *Images* property, such as `ImageList1`.
5. Use the Action Manager editor to add actions to the Action Manager. You can associate an image with an action by setting its *ImageIndex* property to its number in the image list.
6. Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar.
7. For toolbars where you only want to display the icon and no caption: select the Toolbar action band and double-click its Items property. In the collection editor, you can select one or more items and set their *Caption* properties.
8. The images automatically appear on the menu or toolbar.

See Also

TImageList

TCustomActionMainMenuBar

TActionToolBar

3.2.4.5.4 Adding Images to Menu Items

Images can help users navigate in menus by matching glyphs and images to menu item action, similar to toolbar images. You can add single bitmaps to menu items, or you can organize images for your application into an image list and add them to a menu from the image list. If you're using several bitmaps of the same size in your application, it's useful to put them into an image list.

To add a single image to a menu or menu item, set its *Bitmap* property to reference the name of the bitmap to use on the menu or menu item.

To add an image to a menu item using an image list:

1. Drop a *TMainMenu* or *TPopupMenu* object on a form.
2. Drop a *TImageList* object on the form.
3. Open the ImageList editor by double clicking on the *TImageList* object.
4. Click Add to select the bitmap or bitmap group you want to use in the menu. Click OK.
5. Set the *TMainMenu* or *TPopupMenu* object's *Images* property to the ImageList you just created.
6. Create your menu items and submenu items as described in this topic group.
7. Select the menu item you want to have an image in the **Object Inspector** and set the *ImageIndex* property to the corresponding number of the image in the *ImageList* (the default value for *ImageIndex* is -1, which doesn't display an image).

   Note: Use images that are 16 by 16 pixels for proper display in the menu. Although you can use other sizes for the menu images, alignment and consistency problems may result when using images greater than or smaller than 16 by 16 pixels.

See Also

Adding Graphics to Controls (see page 2154)

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.5 Adding, Inserting, and Deleting Menu Items

The following procedures describe how to perform the basic tasks involved in building your menu structure. Each procedure assumes you have the Menu Designer window open.

To add menu items at design time:

1. Select the position where you want to create the menu item.
2. If you've just opened the Menu Designer, the first position on the menu bar is already selected. Begin typing to enter the caption. Or enter the *Name* property first by specifically placing your cursor in the **Object Inspector** and entering a value. In this case, you then need to reselect the *Caption* property and enter a value.
3. Press `Enter`. The next placeholder for a menu item is selected. If you entered the *Caption* property first, use the arrow keys to return to the menu item you just entered. You'll see that Delphi has filled in the *Name* property based on the value you entered for the caption. (See Naming the menu items (see page 1991).)
4. Continue entering values for the *Name* and *Caption* properties for each new item you want to create, or press `Esc` to return to the menu bar. Use the arrow keys to move from the menu bar into the menu, and to then move between items in the list; press `Enter` to complete an action. To return to the menu bar, press `Esc` .

To insert a new, blank menu item:

1. Place the cursor on a menu item.
2. Press `Ins`. Menu items are inserted to the left of the selected item on the menu bar, and above the selected item in the menu list.

To delete a menu item or command:

1. Place the cursor on the menu item you want to delete.
2. Press `Del`.

   Note: You cannot delete the default placeholder that appears below the item last entered in a menu list, or next to the last item on the menu bar. This placeholder does not appear in your menu at runtime.

   Separator bars insert a line between menu items and items on a toolbar. You can use separator bars to indicate groupings within the menu list or toolbar, or simply to provide a visual break in a list.

   To add a separator bar to a menu:

- Add a menu item as described above and type a hyphen (-) for the caption.
- Or press the hyphen (-) key while the cursor is positioned on the menu where you want a separator to appear.

To add a separator bar onto a TActionToolBar, press the insert key and set the new item's caption to a separtor bar (|) or hyphen (-).

To add accelerators or shortcuts to menu items, see Specifying accelerator keys and keyboard shortcuts (see page 2005).

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.6 Adding a Cool Bar Component

Note: The *TCoolBar* component requires version 4.70 or later of COMCTL32.DLL.

The cool bar component (*TCoolBar*)—also called a *rebar*—displays windowed controls on independently movable, resizable

bands. The user can position the bands by dragging the resizing grips on the left side of each band.

To add a cool bar to a form in a VCL application:

1. Add a cool bar component to the form (from the Win32 page of the **Tool palette**). The cool bar automatically aligns to the top of the form.
2. Add windowed controls from the **Tool palette** to the bar.

Only VCL components that descend from *TWinControl* are windowed controls. You can add graphic controls—such as labels or speed buttons—to a cool bar, but they will not appear on separate bands.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.7 Adding a Speed Button to a Panel

To add a speed button to a toolbar panel, place the speed button component (from the Additional category of the **Tool palette**) on the panel.

The panel, rather than the form, "owns" the speed button, so moving or hiding the panel also moves or hides the speed button.

The default height of the panel is 41, and the default height of speed buttons is 25. If you set the *Top* property of each button to 8, they'll be vertically centered. The default grid setting snaps the speed button to that vertical position for you.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.8 Adding a Toolbar Using a Panel Component

To add a toolbar to a form using the panel component

1. Add a panel component to the form (from the Standard category of the **Tool palette**).
2. Set the panel's *Align* property to *alTop*. When aligned to the top of the form, the panel maintains its height, but matches its width to the full width of the form's client area, even if the window changes size.
3. Add speed buttons or other controls to the panel.

Speed buttons are designed to work on toolbar panels. A speed button usually has no caption, only a small graphic (called a *glyph*), which represents the button's function.

Speed buttons have three possible modes of operation. They can

- Act like regular pushbuttons
- Toggle on and off when clicked

- Act like a set of radio buttons

To implement speed buttons on toolbars, do the following:

- Adding a speed button to a panel (see page 1969).
- Assigning a speed button's glyph (see page 1972).
- Setting the initial condition of a speed button (see page 2004).
- Creating a group of speed buttons (see page 1979).
- Allowing toggle buttons (see page 1971).

See Also

TToolBar

TCoolBar

TPanel

TToolButton

3.2.4.5.9 Adding a Toolbar Using the Toolbar Component

The toolbar component (*TToolBar*) offers button management and display features that panel components do not.

To add a toolbar to a form using the toolbar component

1. Add a toolbar component to the form (from the Win32/Common Controls category of the **Tool palette**). The toolbar automatically aligns to the top of the form.
2. Add tool buttons or other controls to the bar.

Tool buttons are designed to work on toolbar components. Like speed buttons, tool buttons can:

- Act like regular pushbuttons.
- Toggle on and off when clicked.
- Act like a set of radio buttons.

To implement tool buttons on a toolbar, do the following:

- Adding a tool button (see page 1971)
- Assigning images to tool buttons (see page 1973)
- Setting tool button appearance and initial conditions (see page 2004)
- Creating groups of tool buttons (see page 1981)
- Allowing toggled tool buttons (see page 1972)

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.10 Adding a Tool Button

To add a tool button to a toolbar, right-click on the toolbar and choose New Button.

The toolbar "owns" the tool button, so moving or hiding the toolbar also moves or hides the button. In addition, all tool buttons on the toolbar automatically maintain the same height and width. You can drop other controls from the **Tool palette** onto the toolbar, and they will automatically maintain a uniform height. Controls will also wrap around and start a new row when they do not fit horizontally on the toolbar.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.11 Adding Hidden Toolbars

Toolbars do not have to be visible all the time. In fact, it is often convenient to have a number of toolbars available, but show them only when the user wants to use them. Often you create a form that has several toolbars, but hide some or all of them.

To create a hidden toolbar:

1. Add a toolbar, cool bar, or panel component to the form.
2. Set the component's *Visible* property to *False*.

Although the toolbar remains visible at design time so you can modify it, it remains hidden at runtime until the application specifically makes it visible.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.12 Allowing Toggle Buttons

Sometimes you want to be able to click a button in a group that's already pressed and have it pop up, leaving no button in the group pressed. Such a button is called a toggle. Use *AllowAllUp* to create a grouped button that acts as a toggle: click it once, it's down; click it again, it pops up.

To make a grouped speed button a toggle, set its *AllowAllUp* property to *True*.

Setting *AllowAllUp* to *True* for any speed button in a group automatically sets the same property value for all buttons in the group. This enables the group to act as a normal group, with only one button pressed at a time, but also allows every button to be up at the same time.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.13 Allowing Toggled Tool Buttons

Use *AllowAllUp* to create a grouped tool button that acts as a toggle: click it once, it is down; click it again, it pops up. To make a grouped tool button a toggle, set its *AllowAllUp* property to *True*.

As with speed buttons, setting *AllowAllUp* to *True* for any tool button in a group automatically sets the same property value for all buttons in the group.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.14 Assigning a Menu to a Tool Button

If you are using a toolbar (*TToolBar*) with tool buttons (*TToolButton*), you can associate menu with a specific button:

To assign a menu to a tool button

1. Select the tool button.
2. In the **Object Inspector**, assign a pop-up menu (*TPopupMenu*) to the tool button's *DropDownMenu* property.

If the menu's *AutoPopup* property is set to *True*, it will appear automatically when the button is pressed.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.15 Assigning a Speed Button's Glyph

Each speed button needs a graphic image called a *glyph* to indicate to the user what the button does. If you supply the speed button only one image, the button manipulates that image to indicate whether the button is pressed, unpressed, selected, or disabled. You can also supply separate, specific images for each state if you prefer.

You normally assign glyphs to speed buttons at design time, although you can assign different glyphs at runtime.

To assign a glyph to a speed button at design time

1. Select the speed button.
2. In the Object Inspector, select the *Glyph* property.
3. Double-click the Value column beside *Glyph* to open the Picture Editor and select the desired bitmap.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.16 Assigning Images to Tool Buttons

Each tool button has an *ImageIndex* property that determines what image appears on it at runtime. If you supply the tool button only one image, the button manipulates that image to indicate whether the button is disabled.

To assign images to tool buttons at design time

1. Select the toolbar on which the buttons appear.
2. In the **Object Inspector**, assign a *TImageList* object to the toolbar's *Images* property. An image list is a collection of same-sized icons or bitmaps.
3. Select a tool button.
4. In the **Object Inspector**, assign an integer to the tool button's *ImageIndex* property that corresponds to the image in the image list that you want to assign to the button.

You can also specify separate images to appear on the tool buttons when they are disabled and when they are under the mouse pointer. To do so, assign separate image lists to the toolbar's *DisabledImages* and *HotImages* properties.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.17 Building Menus

You add a menu component to your form, or forms, for every menu you want to include in your application. You can build each menu structure entirely from scratch, or you can start from one of the predesigned menu templates.

For more information about menu templates, see Using menu templates (see page 2009).

For more information about creating a menu using the menu designer see

- Naming menus (see page 1991)

- Naming the menu items (see page 1991)
- Adding (see page 1968)
- Creating submenus (see page 1978)
- Adding images to menu items (see page 1967)
- Viewing the menu (see page 2011)

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.18 Common Controls and XP Themes

Microsoft has forked Windows common controls into two separate versions. Version 5 is available on all Windows versions from Windows 95 or later; it displays controls using a "3D chiseled" look. Version 6 became available with Windows XP. Under version 6, controls are rendered by a theme engine which matches the current Windows XP theme. If the user changes the theme, version 6 common controls will match the new theme automatically. You don't need to recompile the application.

The VCL can now accommodate both types of common controls. CodeGear has added a number of components to the VCL to handle common control issues automatically and transparently. These components will be present in any VCL application you build. By default, any VCL applications will display version 5 common controls. To display version 6 controls, you (or your application's users) must add a manifest file to your application.

A manifest file contains an XML list of dependencies for your application. The file itself shares the name of your application, with ".manifest" appended to the end. For example, if your project creates Project1.exe as its executable, its manifest file should be named Project1.exe.manifest. Here is an example of a manifest file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
version="1.0.0.0"
processorArchitecture="X86"
name="CompanyName.ProductName.YourApp"
type="win32"
/>
<description>Your application description here.</description>
<dependency>
<dependentAssembly>
<assemblyIdentity
type="win32"
name="Microsoft.Windows.Common-Controls"
version="6.0.0.0"
processorArchitecture="X86"
publicKeyToken="6595b64144ccf1df"
language="*"
/>
</dependentAssembly>
</dependency>
</assembly>
```

Use the example above to create a manifest file for your application. If you place your manifest file in the same directory as your application, its controls will be rendered using the common controls version 6 theme engine. Your application now supports Windows XP themes.

For more information on Windows XP common controls, themes, and manifest files, consult Microsoft's online documentation.

3.2.4.5.19 Controlling When Forms Reside in Memory

By default, Delphi automatically creates the application's main form in memory by including the following code in the application's main entry point:

```
Application.CreateForm(TForm1, Form1);
Application ->CreateForm(__classid(TForm1), &Form1);
```

This function creates a global variable with the same name as the form. So, every form in an application has an associated global variable. This variable is a pointer to an instance of the form's class and is used to reference the form while the application is running. Any unit that includes the form's unit in its **uses** clause can access the form via this variable.

All forms created in this way in the project unit appear when the program is invoked and exist in memory for the duration of the application.

See Also

TForm

3.2.4.5.20 Creating and Managing Menus

Menus provide an easy way for your users to execute logically grouped commands. The Menu Designer enables you to easily add a menu—either predesigned or custom tailored—to your form. You add a menu component to the form, open the Menu Designer, and type menu items directly into the Menu Designer window. You can add or delete menu items, or drag and drop them to rearrange them during design time.

You don't even need to run your program to see the results—your design is immediately visible in the form, appearing just as it will during runtime. Your code can also change menus at runtime, to provide more information or options to the user.

This topic explains how to use the Menu Designer to design menu bars and pop-up (local) menus. It discusses the following ways to work with menus at design time and runtime:

- Opening the Menu Designer (see page 1992).
- Building menus (see page 1973).
- Editing menu items in the Object Inspector (see page 1983).
- Using the Menu Designer context menu (see page 2010).
- Using menu templates (see page 2009).
- Saving a menu as a template (see page 2001).
- Adding images to menu items (see page 1967).

For information about hooking up menu items to the code that executes when they are selected, see .

See Also

Associating Menu Events with Event Handlers (see page 2144)

3.2.4.5.21 Creating and Using Component Templates

You can create templates that are made up of one or more components. After arranging components on a form, setting their properties, and writing code for them, save them as a *component template*. Later, by selecting the template from the **Tool palette**, you can place the preconfigured components on a form in a single step; all associated properties and event-handling code are added to your project at the same time.

Once you place a template on a form, you can reposition the components independently, reset their properties, and create or modify event handlers for them just as if you had placed each component in a separate operation.

To create a component template:

1. Place and arrange components on a form. In the **Object Inspector**, set their properties and events as desired.
2. Select the components. The easiest way to select several components is to drag the mouse over all of them. Gray handles appear at the corners of each selected component.
3. Choose **Component ▸ Create Component Template**.
4. Specify a name for the component template in the Component Template Information edit box. The default proposal is the component type of the first component selected in step 2 followed by the word "Template." For example, if you select a label and then an edit box, the proposed name will be "TLabelTemplate." You can change this name, but be careful not to duplicate existing component names.
5. In the Palette page edit box, specify the **Tool palette** page where you want the template to reside. If you specify a page that does not exist, a new page is created when you save the template.
6. Next to Palette Icon, select a bitmap to represent the template on the palette. The default proposal will be the bitmap used by the component type of the first component selected in step 2. To browse for other bitmaps, click Change. The bitmap you choose must be no larger than 24 pixels by 24 pixels.
7. Click OK.

To remove templates from the **Tool palette**, choose **Component ▸ Configure Palette**.

See Also

Overview of Component Creation (see page 1313)

Working with Frames (see page 2012)

Reusing Components and Groups of Components (see page 2000)

3.2.4.5.22 Creating Customizable Toolbars and Menus

You can use action bands with the Action Manager to create customizable toolbars and menus. At runtime, users of your application can customize the toolbars and menus (action bands) in the application user interface using a customization dialog similar to the Action Manager editor.

To allow the user of your application to customize an action band in your application:

1. Drop an Action Manager component onto a form.
2. Drop your action band components (TCustomActionMainMenuBar,TActionToolBar).
3. Double-click the Action Manager to display the Action Manager editor:

- Add the actions you want to use in your application. Also add the Customize action, which appears at the bottom of the standard actions list.
- Drop a *TCustomizeDlg* component from the Additional tab onto the form, and connect it to the Action Manager using its ActionManager property. You specify a filename for where to stream customizations made by users.
- Drag and drop the actions onto the action band components. (Make sure you add the Customize action to the toolbar or menu.)

4. Complete your application.

When you compile and run the application, users can access a Customize command that displays a customization dialog box similar to the Action Manager editor. They can drag and drop menu items and create toolbars using the same actions you supplied in the Action Manager.

See Also

TActionManager

TActionMainMenuBar

TActionToolBar

TCustomizeDlg

3.2.4.5.23 Creating Dynamic Menus

Dynamic menus and toolbars allow users to modify the application in various ways at run time. Some examples of dynamic usage include customizing the appearance of toolbars and menus, hiding unused items, and responding to most recently used lists (MRUs).

See Also

TActionManager

TActionBars

TActionClients

TActionClientItem

3.2.4.5.24 Creating Frames

To create an empty frame, choose **File** ▸ **New** ▸ **Delphi Projects** ▸ **Delphi Files** and double-click Frame. You can then drop components (including other frames) onto your new frame.

It is usually best—though not necessary—to save frames as part of a project. If you want to create a project that contains only frames and no forms, choose **File** ▸ **New** ▸ **Application**, close the new form and unit without saving them, then choose **File** ▸ **New** ▸ **Delphi Projects** ▸ **Delphi Files** ▸ **Frame** and save the project.

Note: When you save frames, avoid using the default names *Unit1*, *Project1*, and so forth, since these are likely to cause conflicts when you try to use the frames later.

At design time, you can display any frame included in the current project by choosing **View** ▸ **Forms** and selecting a frame. As with forms and data modules, you can toggle between the Form Designer and the frame's form file by right-clicking and choosing View as Form or View as Text.

Adding frames to the Tool palette

Frames are added to the **Tool palette** as component templates. To add a frame to the **Tool palette**, open the frame in the Form Designer (you cannot use a frame embedded in another component for this purpose), right-click the frame, and choose Add to Palette. When the Component Template Information dialog opens, select a name, palette page, and icon for the new template.

See Also

Working with Frames (see page 2012)

Using and Modifying Frames (see page 2008)

Sharing Frames (see page 2005)

Creating and Using Component Templates (see page 1975)

3.2.4.5.25 Creating Most Recently Used Lists

A most recently used list (MRU) reflects the user's most recently accessed files in a specific application. Using action bands, you can code MRU lists in your applications.

When building MRUs for your applications, it is important not to hard code references to specific numerical indexes into the Action Manager's *ActionBars* property. At runtime, the user may change the order of items or even delete them from the action

bands, which in turn will change the numerical ordering of the index. Instead of referring to index numbering, *TActionManager* includes methods that facilitate finding items by action or by caption.

For more information about MRU lists, sample code, and methods for finding actions in lists, see FindItemByAction and FindItemByCaption in the online Help.

See Also

TActionManager

TActionBars

TActionClients

TActionClientItem

3.2.4.5.26 Creating Submenus

Many application menus contain drop-down lists that appear next to a menu item to provide additional, related commands. Such lists are indicated by an arrow to the right of the menu item. Delphi supports as many levels of such submenus as you want to build into your menu.

Organizing your menu structure this way can save vertical screen space. However, for optimal design purposes you probably want to use no more than two or three menu levels in your interface design. (For pop-up menus, you might want to use only one submenu, if any.)

To create a submenu:

1. Select the menu item under which you want to create a submenu.
2. Press `Ctrl+RIGHT ARROW` to create the first placeholder, or right-click and choose Create Submenu.
3. Type a name for the submenu item, or drag an existing menu item into this placeholder.
4. Press `ENTER`, or `DOWN ARROW`, to create the next placeholder.
5. Repeat steps 3 and 4 for each item you want to create in the submenu.
6. Press `ESC` to return to the previous menu level.

Creating submenus by demoting existing menus

You can create a submenu by inserting a menu item from the menu bar (or a menu template) between menu items in a list. When you move a menu into an existing menu structure, all its associated items move with it, creating a fully intact submenu. This pertains to submenus as well. Moving a menu item into an existing submenu just creates one more level of nesting.

See Also

Creating and Managing Menus (see page 1975)

Moving Menu Items (see page 1990)

3.2.4.5.27 Creating Toolbars and Menus

Note: This topic describes the recommended method for creating menus and toolbars in Windows applications. For cross-platform development, you need to use *TToolBar* and the menu components, such as *TMainMenu,* organizing them using action lists (*TActionList*). See Setting up action lists (see page 2003) for details.

You use the Action Manager to automatically generate toolbars and main menus based on the actions contained in your application. The Action Manager manages standard actions and any custom actions that you have written. You then create UI elements based on these actions and use action bands to render the actions items as either menu items or as buttons on a toolbar.

The general procedure for creating menus, toolbars, and other action bands involves these steps:

- Drop an Action Manager onto a form.
- Add actions to the Action Manager, which organizes them into appropriate action lists.
- Create the action bands (that is, the menu or the toolbar) for the user interface.
- Drag and drop the actions into the application interface.

The following procedure explains these steps in more detail.

To create menus and toolbars using action bands:

1. From the Additional category of the **Tool palette**, drop an Action Manager component (TActionManager) onto the form where you want to create the toolbar or menu.
2. If you want images on the menu or toolbar, drop an ImageList component from the Win32 category of the **Tool palette** onto a form. (You need to add the images you want to use to the ImageList or use the one provided.)
3. From the Additional category of the **Tool palette**, drop one or more of the following action bands onto the form:

- TCustomActionMainMenuBar*(for designing main menus)*
- *TActionToolBar(for designing toolbars)*

4. Connect the ImageList to the Action Manager: with focus on the Action Manager and in the **Object Inspector**, select the name of the ImageList from the Images property.
5. Add actions to the Action Manager editor's action pane:

- Double-click the Action Manager to display the Action Manager editor.
- Click the drop-down arrow next to the New Action button (the leftmost button at the top right corner of the Actions tab) and select New Action or New Standard Action. A tree view is displayed. Add one or more actions or categories of actions to the Action Manager's actions pane. The Action Manager adds the actions to its action lists.

6. Drag and drop single actions or categories of actions from the Action Manager editor onto the menu or toolbar you are designing.

To add user-defined actions, create a new *TAction* by pressing the New Action button and writing an event handler that defines how it will respond when fired. See What happens when an action fires (see page 1984) for details. Once you've defined the actions, you can drag and drop them onto menus or toolbars like the standard actions.

See Also

TActionManager

TActionList

TAction

TActionLink

TActionMainMenuBar

TActionToolBar

Setting Up Action Lists (see page 2003)

3.2.4.5.28 Creating a Group of Speed Buttons

A series of speed buttons often represents a set of mutually exclusive choices. In that case, you need to associate the buttons into a group, so that clicking any button in the group causes the others in the group to pop up.

To associate any number of speed buttons into a group, assign the same number to each speed button's *GroupIndex* property.

The easiest way to do this is to select all the buttons you want in the group, and, with the whole group selected, set *GroupIndex* to a unique value.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.29 Creating Forms Dynamically

You may not always want all your application's forms in memory at once. To reduce the amount of memory required at load time, you may want to create some forms only when you need to use them. For example, a dialog box needs to be in memory only during the time a user interacts with it.

To create a form at a different stage during execution using the IDE:

1. Select the **File** ▸ **New** ▸ **Form** from the main menu to display the new form.
2. Remove the form from the Auto-create forms list of the **Project** ▸ **Options** ▸ **Forms** page. This removes the form's invocation at startup. As an alternative, you can manually remove the following line from program's main entry point:

```
Application.CreateForm(TResultsForm, ResultsForm);
Application->CreateForm(__classid(TResultsForm), &ResultsForm);
```

3. Invoke the form when desired by using the form's *Show* method, if the form is modeless, or *ShowModal* method, if the form is modal.

An event handler for the main form must create an instance of the result form and destroy it. One way to invoke the result form is to use the global variable as follows. Note that *ResultsForm* is a modal form so the handler uses the *ShowModal* method.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  ResultsForm := TResultForm.Create(self);
  try
    ResultsForm.ShowModal;
  finally
    ResultsForm.Free;
  end;
end;
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
ResultsForm = new TResultsForm(this);
ResultsForm->ShowModal();
delete ResultsForm;
}
```

In the above example, note the use of **try..finally**. Putting in the line ResultsForm.Free; in the **finally** clause ensures that the memory for the form is freed even if the form raises an exception.

The event handler in the example deletes the form after it is closed, so the form would need to be recreated if you needed to use *ResultsForm* elsewhere in the application. If the form were displayed using *Show* you could not delete the form within the event handler because *Show* returns while the form is still open.

Note: If you create a form using its constructor, be sure to check that the form is not in the Auto-create forms list on the Project ->Options->Forms

page. Specifically, if you create the new form without deleting the form of the same name from the list, Delphi creates the form at startup and this event-handler creates a new instance of the form, overwriting the reference to the auto-created instance. The auto-created instance still exists, but the application can no longer access it. After the event-handler terminates, the global

variable no longer points to a valid form. Any attempt to use the global variable will likely crash the application.

See Also

TForm

3.2.4.5.30 Creating Groups of Tool Buttons

To create a group of tool buttons, select the buttons you want to associate and set their *Style* property to *tbsCheck*; then set their *Grouped* property to *True*. Selecting a grouped tool button causes other buttons in the group to pop up, which is helpful to represent a set of mutually exclusive choices.

Any unbroken sequence of adjacent tool buttons with *Style* set to *tbsCheck* and *Grouped* set to *True* forms a single group. To break up a group of tool buttons, separate the buttons with any of the following:

- A tool button whose *Grouped* property is *False*.
- A tool button whose *Style* property is not set to *tbsCheck*. To create spaces or dividers on the toolbar, add a tool button whose *Style* is *tbsSeparator* or *tbsDivider*.
- Another control besides a tool button.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.31 Creating Modeless Forms Such as Windows

You must guarantee that reference variables for modeless forms exist for as long as the form is in use. This means that these variables should have global scope. In most cases, you use the global reference variable that was created when you made the form (the variable name that matches the name property of the form). If your application requires additional instances of the form, declare separate global variables for each instance.

See Also

TForm

3.2.4.5.32 Demo Programs: Actions, Action Lists, Menus, and Toolbars

For examples of Windows applications that use actions, action lists, menus, and toolbars, refer to <InstallDrive>\Documents and Settings\All Users\Documents\RAD Studio\6.0\Demos\DelphiWin32\VCLWin32\RichEdit (Windows XP), <InstallDrive>\Users\All Users\Documents\RAD Studio\6.0\Demos\DelphiWin32\VCLWin32\RichEdit (Windows Vista). In addition, the Application wizard (**File ▶ New ▶ Other**), MDI Application, SDI Application, and Winx Logo Applications can use the action and action list objects.

See Also

TActionList

TAction

TActionLink

3.2.4.5.33 Designing Toolbars and Cool Bars

A *toolbar* is a panel, usually across the top of a form (under the menu bar), that holds buttons and other controls. A *cool bar* (also called a rebar) is a kind of toolbar that displays controls on movable, resizable bands. If you have multiple panels aligned to the top of the form, they stack vertically in the order added.

You can put controls of any sort on a toolbar. In addition to buttons, you may want to put use color grids, scroll bars, labels, and so on.

You can add a toolbar to a form in several ways:

- Place a panel (*TPanel*) on the form and add controls (typically speed buttons) to it.
- Use a toolbar component (*TToolBar*) instead of *TPanel*, and add controls to it. *TToolBar* manages buttons and other controls, arranging them in rows and automatically adjusting their sizes and positions. If you use tool button (*TToolButton*) controls on the toolbar, *TToolBar* makes it easy to group the buttons functionally and provides other display options.
- Use a cool bar (*TCoolBar*) component and add controls to it. The cool bar displays controls on independently movable and resizable bands.

How you implement your toolbar depends on your application. The advantage of using the Panel component is that you have total control over the look and feel of the toolbar.

By using the toolbar and cool bar components, you are ensuring that your application has the look and feel of a Windows application because you are using the native Windows controls. If these operating system controls change in the future, your application could change as well. Also, since the toolbar and cool bar rely on common components in Windows, your application requires the COMCTL32.DLL. Toolbars and cool bars are not supported in WinNT 3.51 applications.

The following sections describe how to:

- Adding a toolbar using a panel component (see page 1969).
- Adding a toolbar using the toolbar component (see page 1970).
- Adding a cool bar component (see page 1968).
- Responding to clicks (see page 1996).
- Adding hidden toolbars (see page 1971).
- Hiding and showing toolbars (see page 1987).

See Also

TToolBar

TCoolBar

TPanel

TToolButton

3.2.4.5.34 Determining the Order of Merged Menu Items: GroupIndex Property

The *GroupIndex* property determines the order in which the merging menu items appear in the shared menu bar. Merging menu items can replace those on the main menu bar, or can be inserted.

The default value for *GroupIndex* is 0. Several rules apply when specifying a value for *GroupIndex*:

Rules	Description
Lower numbers appear first (farther left) in the menu.	For instance, set the *GroupIndex* property to 0 (zero) for a menu that you always want to appear leftmost, such as a File menu. Similarly, specify a high number (it needn't be in sequence) for a menu that you always want to appear rightmost, such as a Help menu.

To replace items in the main menu, give items on the child menu the same *GroupIndex* value.	This can apply to groupings or to single items. For example, if your main form has an Edit menu item with a *GroupIndex* value of 1, you can replace it with one or more items from the child form's menu by giving them a *GroupIndex* value of 1 as well. Giving multiple items in the child menu the same *GroupIndex* value keeps their order intact when they merge into the main menu.
To insert items without replacing items in the main menu, leave room in the numeric range of the main menu's items and "plug in" numbers from the child form.	For example, number the items in the main menu 0 and 5, and insert items from the child menu by numbering them 1, 2, 3, and 4.

See Also

TMenu

3.2.4.5.35 Developing the Application User Interface: Overview

When you open the IDE or create a new project, a blank form is displayed on the screen. You design your application's user interface (UI) by placing and arranging visual components, such as windows, menus, and dialog boxes, from the **Tool palette** onto the form.

Once a visual component is on the form, you can adjust its position, size, and other design-time properties, and code its event handlers (see page 2149). The form takes care of the underlying programming details.

The following topics describe some of the major interface tasks, such as working with forms, creating component templates, adding dialog boxes, and organizing actions for menus and toolbars.

3.2.4.5.36 Displaying an Auto-created Form

If you choose to create a form at startup, and do not want it displayed until sometime later during program execution, the form's event handler uses the *ShowModal* method to display the form that is already loaded in memory:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
ResultsForm.ShowModal;
end;
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
ResultsForm->ShowModal();
}
```

In this case, since the form is already in memory, there is no need to create another instance or destroy that instance.

See Also

TForm

3.2.4.5.37 Editing Menu Items in the Object Inspector

This topic has discussed how to set several properties for menu items—for example, the *Name* and *Caption* properties—by using the Menu Designer.

The section has also described how to set menu item properties, such as the *ShortCut* property, directly in the **Object Inspector**, just as you would for any component selected in the form.

When you edit a menu item by using the Menu Designer, its properties are still displayed in the **Object Inspector**. You can

switch focus to the **Object Inspector** and continue editing the menu item properties there. Or you can select the menu item from the Component list in the **Object Inspector** and edit its properties without ever opening the Menu Designer.

To close the Menu Designer window and continue editing menu items:

1. Switch focus from the Menu Designer window to the **Object Inspector** by clicking the properties page of the **Object Inspector**.
2. Close the Menu Designer as you normally would. The focus remains in the **Object Inspector**, where you can continue editing properties for the selected menu item. To edit another menu item, select it from the Component list.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.38 What Happens When an Action Fires

When an event fires, a series of events intended primarily for generic actions occurs. Then if the event doesn't handle the action, another sequence of events occurs.

Responding with events

When a client component or control is clicked or otherwise acted on, a series of events occurs to which you can respond. For example, the following code illustrates the event handler for an action that toggles the visibility of a toolbar when the action is executed:

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
{ Toggle Toolbar1's visibility }
ToolBar1.Visible := not ToolBar1.Visible;
end;
void __fastcall TForm1::Action1Execute(TObject *Sender)
{
// Toggle Toolbar1's visibility
ToolBar1->Visible = !ToolBar1->Visible;
}
```

Note: For general information about events and event handlers, see Working with Events and Event Handlers (see page 2149).

You can supply an event handler that responds at one of three different levels: the action, the action list, or the application. This is only a concern if you are using a new generic action rather than a predefined standard action. You do not have to worry about this if using the standard actions because standard actions have built-in behavior that executes when these events occur.

The order in which the event handlers will respond to events is as follows:

- Action list
- Application
- Action

When the user clicks on a client control, Delphi calls the action's Execute method which defers first to the action list, then the Application object, then the action itself if neither action list nor Application handles it. To explain this in more detail, Delphi follows this dispatching sequence when looking for a way to respond to the user action:

If you supply an *OnExecute* event handler for the action list and it handles the action, the application proceeds.

The action list's event handler has a parameter called Handled, that returns False by default. If the handler is assigned and it handles the event, it returns *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ActionList1ExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
```

```
Handled := True;
end;
void __fastcall TForm1::ApplicationExecuteAction(TBasicAction *Action, bool &Handled)
{
  // Prevent execution of all actions in Application
Handled = true;
}
void __fastcall TForm1::ActionList1ExecuteAction(TBasicAction *Action, bool &Handled)
{
Handled = true;
}
```

If you don't set Handled to *True* in the action list event handler, then processing continues.

If you did not write an *OnExecute* event handler for the action list or if the event handler doesn't handle the action, the application's *OnActionExecute* event handler fires. If it handles the action, the application proceeds.

The global Application object receives an OnActionExecute event if any action list in the application fails to handle an event. Like the action list's *OnExecute* event handler, the *OnActionExecute* handler has a parameter Handled that returns *False* by default. If an event handler is assigned and handles the event, it returns *True*, and the processing sequence ends here. For example:

```
procedure TForm1.ApplicationExecuteAction(Action: TBasicAction; var Handled: Boolean);
begin
{ Prevent execution of all actions in Application }
Handled := True;
end;
```

If the application's *OnExecute* event handler doesn't handle the action, the action's *OnExecute* event handler fires.

You can use built-in actions or create your own action classes that know how to operate on specific target classes (such as edit controls). When no event handler is found at any level, the application next tries to find a target on which to execute the action. When the application locates a target that the action knows how to address, it invokes the action. See how actions find their targets (see page 1987) for details on how the application locates a target that can respond to a predefined action class.

See Also

TActionList

TAction

TActionLink

Setting Up Action Lists (see page 2003)

Using Action Lists (see page 2007)

3.2.4.5.39 Handling the Screen

A global variable of type *TScreen* called *Screen* is created when you create a project. Screen encapsulates the state of the screen on which your application is running. Common tasks performed by *Screen* include specifying:

- The look of the cursor.
- The size of the window in which your application is running.
- A list of fonts available to the screen device.
- Multiple screen behavior (Windows only).

If your Windows application runs on multiple monitors, *Screen* maintains a list of monitors and their dimensions so that you can effectively manage the layout of your user interface.

See Also

TScreen

3.2.4.5.40 Hiding the Main Form

You can prevent the main form from appearing when your application starts by using the global Application variable (see page 2012).

To hide the main form at startup:

1. Choose **Project** ▸ **View Source** to display the main project file.
2. Add the following code after the call to Application.CreateForm and before the call to Application.Run.

```
Application.ShowMainForm := False;
Form1.Visible := False; { the name of your main form may differ }
Application->ShowMainForm = false;
```

Note: You can set the form's *Visible* property to *False* using the Object Inspector

at design time rather than setting it at runtime as in the previous example.

See Also

ShowMainForm

Application

Using the Main Form (see page 2009)

3.2.4.5.41 Hiding Unused Items and Categories in Action Bands

One benefit of using ActionBands is that unused items and categories can be hidden from the user. Over time, the action bands become customized for the application users, showing only the items that they use and hiding the rest from view. Hidden items can become visible again when the user presses a drop-down button. Also, the user can restore the visibility of all action band items by resetting the usage statistics from the customization dialog. Item hiding is the default behavior of action bands, but that behavior can be changed to prevent hiding of individual items, all the items in a particular collection (like the File menu), or all of the items in a given action band.

The action manager keeps track of the number of times an action has been called by the user, which is stored in the associated *TActionClientItem's UsageCount* field. The action manager also records the number of times the application has been run, which we shall call the session number, as well as the session number of the last time an action was used. The value of *UsageCount* is used to look up the maximum number of sessions the item can go unused before it becomes hidden, which is then compared with the difference between the current session number and the session number of the last use of the item. If that difference is greater than the number determined in *PrioritySchedule*, the item is hidden. The default values of *PrioritySchedule* are shown in the table below:

Default values of the action manager's PrioritySchedule property

Number of sessions in which an action band item was used	Number of sessions an item will remain unhidden after its last use
0, 1	3
2	6
3	9
4, 5	12
6-8	17
9-13	23
14-24	29

25 or more	31

It is possible to disable item hiding at design time. To prevent a specific action (and all the collections containing it) from becoming hidden, find its *TActionClientItem* object and set its *UsageCount* to -1. To prevent hiding for an entire collection of items, such as the File menu or even the main menu bar, find the *TActionClients* object associated with the collection and set its *HideUnused* property to *False*.

See Also

TActionManager

TActionBars

TActionClients

TActionClientItem

3.2.4.5.42 Hiding and Showing Toolbars

Often, you want an application to have multiple toolbars, but you do not want to clutter the form with them all at once. Or you may want to let users decide whether to display toolbars. As with all components, toolbars can be shown or hidden at runtime as needed.

To show or hide a toolbar at runtime, set its *Visible* property to *False* or *True*, respectively. Usually you do this in response to particular user events or changes in the operating mode of the application. To do this, you typically have a close button on each toolbar. When the user clicks that button, the application hides the corresponding toolbar.

You can also provide a means of toggling the toolbar. In the following example, a toolbar of pens is toggled from a button on the main toolbar. Since each click presses or releases the button, an *OnClick* event handler can show or hide the Pen toolbar depending on whether the button is up or down.

```
procedure TForm1.PenButtonClick(Sender: TObject);
begin
  PenBar.Visible := PenButton.Down;
end;
void __fastcall TForm1::PenButtonClick(TObject *Sender)
{
  PenBar->Visible = PenButton->Down;
}
```

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.43 How Actions Find Their Targets

What happens when an action fires (see page 1984) describes the execution cycle that occurs when a user invokes an action. If no event handler is assigned to respond to the action, either at the action list, application, or action level, then the application tries to identify a target object to which the action can apply itself.

The application looks for the target using the following sequence:

1. Active control: The application looks first for an active control as a potential target.
2. Active form: If the application does not find an active control or if the active control can't act as a target, it looks at the screen's ActiveForm.
3. Controls on the form: If the active form is not an appropriate target, the application looks at the other controls on the active form for a target.

If no target is located, nothing happens when the event is fired.

Some controls can expand the search to defer the target to an associated component; for example, data-aware controls defer to the associated dataset component. Also, some predefined actions do not use a target; for example, the File Open dialog.

See Also

TActionList

TAction

TActionLink

3.2.4.5.44 Developing Dialog Boxes

The dialog box components on the Dialogs category of the **Tool palette** make various dialog boxes available to your applications. These dialog boxes provide applications with a familiar, consistent interface that enables the user to perform common file operations such as opening, saving, and printing files. Dialog boxes display and/or obtain data.

Each dialog box opens when its *Execute* method is called. *Execute* returns a Boolean value: if the user chooses OK to accept any changes made in the dialog box, *Execute* returns *True*; if the user chooses Cancel to escape from the dialog box without making or saving changes, *Execute* returns *False*.

3.2.4.5.45 Importing Resource Files

You can build menus with other applications, so long as the menus are in the standard Windows resource (.RC) file format. You can import such menus directly into your project, saving you the time and effort of rebuilding menus that you created elsewhere.

To load existing .RC menu files

1. In the Menu Designer, place your cursor where you want the menu to appear. The imported menu can be part of a menu you are designing, or an entire menu in itself.
2. Right-click and choose Insert From Resource. The Insert Menu From Resource dialog box appears.
3. In the dialog box, select the resource file you want to load, and choose OK. The menu appears in the Menu Designer window.

   Note: If your resource file contains more than one menu, you first need to save each menu as a separate resource file before importing it.

See Also

TMenu

3.2.4.5.46 Managing Layout

At its simplest, you control the layout of your user interface by where you place controls in your forms. The placement choices you make are reflected in the control's *Top*, *Left*, *Width*, and *Height* properties. You can change these values at runtime to change the position and size of the controls in your forms.

Controls have a number of other properties, however, that allow them to automatically adjust to their contents or containers. This allows you to lay out your forms so that the pieces fit together into a unified whole.

Two properties affect how a control is positioned and sized in relation to its parent. The *Align* property lets you force a control to fit perfectly within its parent along a specific edge or filling up the entire client area after any other controls have been aligned. When the parent is resized, the controls aligned to it are automatically resized and remain positioned so that they fit against a particular edge.

If you want to keep a control positioned relative to a particular edge of its parent, but don't want it to necessarily touch that edge or be resized so that it always runs along the entire edge, you can use the *Anchors* property.

If you want to ensure that a control does not grow too big or too small, you can use the *Constraints* property. *Constraints* lets you specify the control's maximum height, minimum height, maximum width, and minimum width. Set these to limit the size (in pixels) of the control's height and width. For example, by setting the *MinWidth* and *MinHeight* of the constraints on a container object, you can ensure that child objects are always visible.

The value of *Constraints* propagates through the parent/child hierarchy so that an object's size can be constrained because it contains aligned children that have size constraints. *Constraints* can also prevent a control from being scaled in a particular dimension when its *ChangeScale* method is called.

TControl introduces a protected event, *OnConstrainedResize*, of type *TConstrainedResizeEvent*:

```
TConstrainedResizeEvent = procedure(Sender: TObject; var MinWidth, MinHeight, MaxWidth,
MaxHeight: Integer) of object;
void __fastcall (__closure *TConstrainedResizeEvent)(System::TObject* Sender, int &MinWidth,
int &MinHeight, int &MaxWidth, int &MaxHeight);
```

This event allows you to override the size constraints when an attempt is made to resize the control. The values of the constraints are passed as var parameters which can be changed inside the event handler. *OnConstrainedResize* is published for container objects (*TForm*, *TScrollBox*, *TControlBar*, and *TPanel*). In addition, component writers can use or publish this event for any descendant of *TControl*.

Controls that have contents that can change in size have an *AutoSize* property that causes the control to adjust its size to its font or contained objects.

3.2.4.5.47 Manipulating Menu Items at Runtime

Sometimes you want to add menu items to an existing menu structure while the application is running, to provide more information or options to the user. You can insert a menu item by using the menu item's *Add* or *Insert* method, or you can alternately hide and show the items in a menu by changing their *Visible* property. The *Visible* property determines whether the menu item is displayed in the menu. To dim a menu item without hiding it, use the Enabled property.

For examples that use the menu item's *Visible* and *Enabled* properties, see Disabling menu items (see page 2160).

In multiple document interface (MDI) and Object Linking and Embedding (OLE) applications, you can also merge menu items into an existing menu bar. See Merging menus (see page 1989) for more information.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.48 Merging Menus

For MDI applications, such as the text editor sample application, and for OLE client applications, your application's main menu needs to be able to receive menu items either from another form or from the OLE server object. This is often called *merging menus*. Note that OLE technology is limited to Windows applications only and is not available for use in cross-platform programming.

You prepare menus for merging by specifying values for two properties:

- *Menu*, a property of the form
- *GroupIndex*, a property of menu items in the menu

See Also

TMenu

3.2.4.5.49 Using Forms

When you create a form from the IDE, Delphi automatically creates the form in memory by including code in the main entry point of your application function. Usually, this is the desired behavior and you don't have to do anything to change it. That is, the main window persists through the duration of your program, so you would likely not change the default behavior when creating the form for your main window.

However, you may not want all your application's forms in memory for the duration of the program execution. That is, if you do not want all your application's dialogs in memory at once, you can create the dialogs dynamically when you want them to appear.

Forms can be modal or modeless. Modal forms are forms with which the user must interact before switching to another form (for example, a dialog box requiring user input). Modeless forms are windows that are displayed until they are either obscured by another window or until they are closed or minimized by the user.

See Also

TForm

3.2.4.5.50 Moving Menu Items

During design time, you can move menu items simply by dragging and dropping. You can move menu items along the menu bar, or to a different place in the menu list, or into a *different* menu entirely.

The only exception to this is hierarchical: you cannot demote a menu item from the menu bar into its own menu; nor can you move a menu item into its own submenu. However, you can move any item into a different menu, no matter what its original position is.

While you are dragging, the cursor changes shape to indicate whether you can release the menu item at the new location. When you move a menu item, any items beneath it move as well.

To move a menu item along the menu bar:

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new location.
2. Release the mouse button to drop the menu item at the new location.

To move a menu item into a menu list:

1. Drag the menu item along the menu bar until the arrow tip of the drag cursor points to the new menu. This causes the menu to open, enabling you to drag the item to its new location.
2. Drag the menu item into the list, releasing the mouse button to drop the menu item at the new location.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.51 Naming Conventions for Template Menu Items and Event Handlers

When you save a menu as a template, Delphi does not save its *Name* property, since every menu must have a unique name

within the scope of its owner (the form). However, when you insert the menu as a template into a new form by using the Menu Designer, Delphi then generates new names for it and all of its items.

For example, suppose you save a File menu as a template. In the original menu, you name it *MyFile*. If you insert it as a template into a new menu, Delphi names it *File1*. If you insert it into a menu with an existing menu item named File1, Delphi names it *File2*.

Delphi also does not save any *OnClick* event handlers associated with a menu saved as a template, since there is no way to test whether the code would be applicable in the new form. When you generate a new event handler for the menu template item, Delphi still generates the event handler name. You can easily associate items in the menu template with existing *OnClick* event handlers in the form.

For more information, see Associating menu events with event handlers (see page 2144).

See Also

TMenu

3.2.4.5.52 Naming Menus

As with all components, when you add a menu component to the form, the form gives it a default name; for example, *MainMenu1*. You can give the menu a more meaningful name that follows language naming conventions.

The menu name is added to the form's type declaration, and the menu name then appears in the Component list.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.53 Naming the Menu Items

In contrast to the menu component itself, you need to explicitly name menu items as you add them to the form. You can do this in one of two ways:

- Directly type the value for the *Name* property.
- Type the value for the *Caption* property first, and let Delphi derive the *Name* property from the caption.

For example, if you give a menu item a *Caption* property value of File, Delphi assigns the menu item a *Name* property of *File1*. If you fill in the *Name* property before filling in the *Caption* property, Delphi leaves the *Caption* property blank until you type a value.

Note: If you enter characters in the *Caption* property that are not valid for Delphi identifiers, Delphi modifies the *Name* property accordingly. For example, if you want the caption to start with a number, Delphi precedes the number with a character to derive the *Name* property.

The following table demonstrates some examples of this, assuming all menu items shown appear in the same menu bar.

Sample captions and their derived names

Component caption	Derived name	Explanation
&File	File1	Removes ampersand
&File (2nd occurrence)	File2	Numerically orders duplicate items
1234	N12341	Adds a preceding letter and numerical order
1234 (2nd occurrence)	N12342	Adds a number to disambiguate the derived name
$@@@#	N1	Removes all non-standard characters, adding preceding letter and numerical order

• (hyphen)	N2	Numerical ordering of second occurrence of caption with no standard characters

As with the menu component, Delphi adds any menu item names to the form's type declaration, and those names then appear in the Component list.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.54 Opening the Menu Designer

You design menus for your application using the Menu Designer. Before you can start using the Menu Designer, first add either a *TMainMenu* or *TPopupMenu* component to your form. Both menu components are located on the Standard category of the **Tool palette**.



A MainMenu component creates a menu that's attached to the form's title bar. A PopupMenu component creates a menu that appears when the user right-clicks in the form. Pop-up menus do not have a menu bar.

To open the Menu Designer, select a menu component on the form, and then either:

- Double-click the menu component.
- Or, from the Properties page of the **Object Inspector**, select the Items property, and then either double-click [Menu] in the Value column, or click the ellipsis (...) button.

The Menu Designer appears, with the first (blank) menu item highlighted in the Designer, and the *Caption* property selected in the **Object Inspector**.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.55 Organizing Actions for Toolbars and Menus

Several features simplify the work of creating, customizing, and maintaining menus and toolbars. These features allow you to organize lists of actions that users of your application can initiate by pressing a button on a toolbar, choosing a command on a menu, or pointing and clicking on an icon.

Often a set of actions is used in more than one user interface element. For example, the Cut, Copy, and Paste commands often appear on both an Edit menu and on a toolbar. You only need to add the action once to use it in multiple UI elements in your application.

On the Windows platform, tools are provided to make it easy to define and group actions, create different layouts, and customize menus at design time or runtime. These tools are known collectively as ActionBand tools, and the menus and toolbars you create with them are known as action bands. In general, you can create an ActionBand user interface as follows:

- Build the action list to create a set of actions that will be available for your application (use the Action Manager, TActionManager)
- Add the user interface elements to the application (use ActionBand components such as TActionMainMenuBarand TActionToolBar)

- Drag-and-drop actions from the Action Manager onto the user interface elements

The following table defines the terminology related to setting up menus and toolbars:

Action setup terminology

Term	Definition
Action	A response to something a user does, such as clicking a menu item. Many standard actions that are frequently required are provided for you to use in your applications as is. For example, file operations such as File Open, File SaveAs, File Run, and File Exit are included along with many others for editing, formatting, searches, help, dialogs, and window actions. You can also program custom actions and access them using action lists and the Action Manager.
Action band	A container for a set of actions associated with a customizable menu or toolbar. The ActionBand components for main menus and toolbars (*TActionMainMenuBar* and *TActionToolBar*) are examples of action bands.
Action category	Lets you group actions and drop them as a group onto a menu or toolbar. For example, one of the standard action categories is Search which includes Find, FindFirst, FindNext, and Replace actions all at once.
Action classes	Classes that perform the actions used in your application. All of the standard actions are defined in action classes such as *TEditCopy*, *TEditCut*, and *TEditUndo*. You can use these classes by dragging and dropping them from the Customize dialog onto an action band.
Action client	Most often represents a menu item or a button that receives a notification to initiate an action. When the client receives a user command (such as a mouse click), it initiates an associated action.
Action list	Maintains a list of actions that your application can take in response to something a user does.
Action Manager	Groups and organizes logical sets of actions that can be reused on ActionBand components. See *TActionManager*.
Menu	Lists commands that the user of the application can execute by clicking on them. You can create menus by using the ActionBand menu class *TActionMainMenuBar*, or by using cross-platform components such as *TMainMenu* or *TPopupMenu*.
Target	Represents the item an action does something to. The target is usually a control, such as a memo or a data control. Not all actions require a target. For example, the standard help actions ignore the target and simply launch the help system.
Toolbar	Displays a visible row of button icons which, when clicked, cause the program to perform some action, such as printing the current document. You can create toolbars by using the ActionBand toolbar component *TActionToolBar*, or by using the cross-platform component *TToolBar*.

If you are doing cross-platform development, refer to Using action lists (see page 2007) for details.

See Also

TActionManager

TActionList

TAction

TActionLink

Setting Up Action Lists (see page 2003)

3.2.4.5.56 Passing Additional Arguments to Forms

Typically, you create forms for your application from within the IDE. When created this way, the forms have a constructor that takes one argument, *Owner*, which is the owner of the form being created. (The owner is the calling application object or form object.) *Owner* can be **nil**.

To pass additional arguments to a form, create a separate constructor and instantiate the form using this new constructor. The

example form class below shows an additional constructor, with the extra argument *whichButton*. This new constructor is added to the form class manually.

```
TResultsForm = class(TForm)
  ResultsLabel: TLabel;
  OKButton: TButton;
  procedure OKButtonClick(Sender: TObject);
private
public
  constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
end;
class TResultsForm : public TForm
{
__published:    // IDE-managed Components
TLabel *ResultsLabel;
TButton *OKButton;
void __fastcall OKButtonClick(TObject *Sender);
private:        // User declarations
public:         // User declarations
virtual __fastcall TResultsForm(TComponent* Owner);
   virtual __fastcall TResultsForm(int whichButton, TComponent* Owner);
};
```

Here's the manually coded constructor that passes the additional argument, *whichButton*. This constructor uses the *whichButton* parameter to set the *Caption* property of a *Label* control on the form.

```
constructor CreateWithButton(whichButton: Integer; Owner: TComponent);
begin
  inherited Create(Owner);
  case whichButton of
    1: ResultsLabel.Caption := "You picked the first button.";
    2: ResultsLabel.Caption := "You picked the second button.";
    3: ResultsLabel.Caption := "You picked the third button.";
  end;
end;
void__fastcall TResultsForm::TResultsForm(int whichButton, TComponent* Owner)
: TForm(Owner)
{
switch (whichButton) {
case 1:
ResultsLabel->Caption = "You picked the first button!";
break;
case 2:
ResultsLabel->Caption = "You picked the second button!";
break;
case 3:
ResultsLabel->Caption = "You picked the third button!";
}
}
```

When creating an instance of a form with multiple constructors, you can select the constructor that best suits your purpose. For example, the following *OnClick* handler for a button on a form calls creates an instance of *TResultsForm* that uses the extra parameter:

```
procedure TMainForm.SecondButtonClick(Sender: TObject);
var
  rf: TResultsForm;
begin
  rf := TResultsForm.CreateWithButton(2, self);
  rf.ShowModal;
  rf.Free;
end;
void __fastcall TMainMForm::SecondButtonClick(TObject *Sender)
{
TResultsForm *rf = new TResultsForm(2, this);
rf->ShowModal();
```

```
delete rf;
}
```

See Also

TForm

3.2.4.5.57 Predefined Action Classes

You can add predefined actions to your application by right-clicking on the Action Manager and choosing New Standard Action. The New Standard Action Classes dialog box is displayed listing the predefined action classes and the associated standard actions. These are actions that are included with Delphi and they are objects that automatically perform actions. The predefined actions are organized within the following classes:

Class	Description
Edit	Standard edit actions: Used with an edit control target. *TEditAction* is the base class for descendants that each override the *ExecuteTarget* method to implement copy, cut, and paste tasks by using the clipboard.
Format	Standard formatting actions: Used with rich text to apply text formatting options such as bold, italic, underline, strikeout, and so on. *TRichEditAction* is the base class for descendants that each override the *ExecuteTarget* and *UpdateTarget* methods to implement formatting of the target.
Help	Standard Help actions: Used with any target. *THelpAction* is the base class for descendants that each override the *ExecuteTarget* method to pass the command onto a Help system.
Window	Standard window actions: Used with forms as targets in an MDI application. *TWindowAction* is the base class for descendants that each override the *ExecuteTarget* method to implement arranging, cascading, closing, tiling, and minimizing MDI child forms.
File	File actions: Used with operations on files such as File Open, File Run, or File Exit.
Search	Search actions: Used with search options. *TSearchAction* implements the common behavior for actions that display a modeless dialog where the user can enter a search string for searching an edit control.
Tab	Tab control actions: Used to move between tabs on a tab control such as the Prev and Next buttons on a wizard.
List	List control actions: Used for managing items in a list view.
Dialog	Dialog actions: Used with dialog components. *TDialogAction* implements the common behavior for actions that display a dialog when executed. Each descendant class represents a specific dialog.
Internet	Internet actions: Used for functions such as Internet browsing, downloading, and sending mail.
DataSet	DataSet actions: Used with a dataset component target. *TDataSetAction* is the base class for descendants that each override the *ExecuteTarget* and *UpdateTarget* methods to implement navigation and editing of the target. *TDataSetAction* introduces a *DataSource* property that ensures actions are performed on that dataset. If *DataSource* is nil, the currently focused data-aware control is used.
Tools	Tools: Additional tools such as *TCustomizeActionBars* for automatically displaying the customization dialog for action bands.

All of the action objects are described under the action object names in the online Help.

See Also

TActionList

TAction

TActionLink

3.2.4.5.58 Registering Actions

When you write your own actions, you can register actions to enable them to appear in the Action List editor. You register and unregister actions by using the global routines in the Actnlist unit:

```
procedure RegisterActions(const CategoryName: string; const AClasses: array of
TBasicActionClass; Resource: TComponentClass);
procedure UnRegisterActions(const AClasses: array of TBasicActionClass);
extern PACKAGE void __fastcall UnRegisterActions(TMetaClass* const * AClasses, const int
AClasses_Size);
extern PACKAGE void __fastcall RegisterActions(const AnsiString CategoryName, TMetaClass*
const * AClasses, const int AClasses_Size, TMetaClass* Resource);
```

When you call RegisterActions, the actions you register appear in the Action List editor for use by your applications. You can supply a category name to organize your actions, as well as a *Resource* parameter that lets you supply default property values.

For example, the following code registers the standard actions with the IDE:

```
{ Standard action registration }
RegisterActions('', [TAction], nil);
RegisterActions('Edit', [TEditCut, TEditCopy, TEditPaste], TStandardActions);
RegisterActions('Window', [TWindowClose, TWindowCascade, TWindowTileHorizontal,
TWindowTileVertical, TWindowMinimizeAll, TWindowArrange], TStandardActions);
namespace MyAction
{
  void __fastcall PACKAGE Register()
  {
     // code goes here to register any components and editors
TMetaClass classes[2] = {__classid(TMyAction1), __classid(TMyAction2)};
RegisterActions("MySpecialActions", classes, 1, NULL);
}
}
```

When you call UnRegisterActions, the actions no longer appear in the Action List editor.

See Also

TActionList

TAction

TActionLink

3.2.4.5.59 Responding to Clicks

When the user clicks a control, such as a button on a toolbar, the application generates an *OnClick* event which you can respond to with an event handler. Since *OnClick* is the default event for buttons, you can generate a skeleton handler for the event by double-clicking the button at design time. For general information about events and event handlers, see Working with Events and Event Handlers (see page 2149) and Generating a handler for a component's default event (see page 2147).

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.60 Retrieving Data from Forms

Most real-world applications consist of several forms. Often, information needs to be passed between these forms. Information can be passed to a form by means of parameters to the receiving form's constructor, or by assigning values to the form's properties. The way you get information from a form depends on whether the form is modal or modeless.

See Also

TForm

3.2.4.5.61 Retrieving Data from Modal Forms

Just like modeless forms, modal forms often contain information needed by other forms. The most common example is when form A launches modal form B. When form B is closed, form A needs to know what the user did with form B to decide how to proceed with the processing of form A. If form B is still in memory, it can be queried through properties or member functions just as in the modeless forms example above. But how do you handle situations where form B is deleted from memory upon closing? Since a form does not have an explicit return value, you must preserve important information from the form before it is destroyed.

To illustrate, consider a modified version of the *ColorForm* form that is designed to be a modal form. The class declaration is as follows:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  SelectButton: TButton;
  CancelButton: TButton;
  procedure CancelButtonClick(Sender: TObject);
  procedure SelectButtonClick(Sender: TObject);
private
  FColor: Pointer;
public
  constructor CreateWithColor(Value: Pointer; Owner: TComponent);
end;
```

```
class TColorForm : public TForm
{
__published:    // IDE-managed Components
TListBox *ColorListBox;
TButton *SelectButton;
TButton *CancelButton;
void __fastcall CancelButtonClick(TObject *Sender);
void __fastcall SelectButtonClick(TObject *Sender);
private:        // User declarations
String* curColor;
public:         // User declarations
virtual __fastcall TColorForm(TComponent* Owner);
virtual __fastcall TColorForm(String* s, TComponent* Owner);
};
```

The form has a listbox called *ColorListBox* with a list of names of colors. When pressed, the button called *SelectButton* makes note of the currently selected color name in *ColorListBox* then closes the form. *CancelButton* is a button that simply closes the form.

Note that a user-defined constructor was added to the class that takes a *Pointer* argument. Presumably, this *Pointer* points to a string that the form launching *ColorForm* knows about. The implementation of this constructor is as follows:

```
constructor TColorForm(Value: Pointer; Owner: TComponent);
begin
  FColor := Value;
  String(FColor^) := '';
end;
```

```
void__fastcall TColorForm::TColorForm(String* s, TComponent* Owner)
: TForm(Owner)
```

```
{
curColor = s;
*curColor = "";
}
```

The constructor saves the pointer to a private data member *FColor* and initializes the string to an empty string.

Note: To use the above user-defined constructor, the form must be explicitly created. It cannot be auto-created when the application is started. For details, see Controlling when forms reside in memory (see page 1975).

In the application, the user selects a color from the listbox and presses *SelectButton* to save the choice and close the form. The *OnClick* event handler for *SelectButton* might look like this:

```
procedure TColorForm.SelectButtonClick(Sender: TObject);
begin
  with ColorListBox do
    if ItemIndex >= 0 then
      String(FColor^) := ColorListBox.Items[ItemIndex];
  end;
  Close;
end;
void __fastcall TColorForm::SelectButtonClick(TObject *Sender)
{
int index = ColorListBox->ItemIndex;
if (index >= 0)
*curColor = ColorListBox->Items->Strings[index];
Close();
}
```

Notice that the event handler stores the selected color name in the string referenced by the pointer that was passed to the constructor.

To use *ColorForm* effectively, the calling form must pass the constructor a pointer to an existing string. For example, assume *ColorForm* was instantiated by a form called *ResultsForm* in response to a button called *UpdateButton* on *ResultsForm* being clicked. The event handler would look as follows:

```
procedure TResultsForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  GetColor(Addr(MainColor));
  if MainColor <> '' then
    {do something with the MainColor string}
  else
    {do something else because no color was picked}
end;
procedure GetColor(PColor: Pointer);
begin
  ColorForm := TColorForm.CreateWithColor(PColor, Self);
  ColorForm.ShowModal;
  ColorForm.Free;
end;
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
String s;
GetColor(&s);
if (s != "") {
// do something with the color name string
}
else {
// do something else because no color was picked
}
}
//---------------------------------------------------------------------
void TResultsForm::GetColor(String *s)
{
```

```
ColorForm = new TColorForm(s, this);
ColorForm->ShowModal();
delete ColorForm;
ColorForm = 0; // NULL the pointer
}
```

UpdateButtonClick creates a String called MainColor. The address of MainColor is passed to the *GetColor* function which creates *ColorForm*, passing the pointer to MainColor as an argument to the constructor. As soon as *ColorForm* is closed it is deleted, but the color name that was selected is still preserved in MainColor, assuming that a color was selected. Otherwise, MainColor contains an empty string which is a clear indication that the user exited *ColorForm* without selecting a color.

This example uses one string variable to hold information from the modal form. Of course, more complex objects can be used depending on the need. Keep in mind that you should always provide a way to let the calling form know if the modal form was closed without making any changes or selections (such as having MainColor default to an empty string).

See Also

TForm

3.2.4.5.62 Retrieving Data from Modeless Forms

You can easily extract information from modeless forms by calling public member functions of the form or by querying properties of the form. For example, assume an application contains a modeless form called *ColorForm* that contains a listbox called *ColorListBox* with a list of colors ("Red," "Green," "Blue," and so on). The selected color name string in *ColorListBox* is automatically stored in a property called *CurrentColor* each time a user selects a new color. The class declaration for the form is as follows:

```
TColorForm = class(TForm)
  ColorListBox:TListBox;
  procedure ColorListBoxClick(Sender: TObject);
private
  FColor:String;
public
  property CurColor:String read FColor write FColor;
end;
class TColorForm : public TForm
{
__published:    // IDE-managed Components
TListBox *ColorListBox;
void __fastcall ColorListBoxClick(TObject *Sender);
private:        // User declarations
String getColor();
void   setColor(String);
String curColor;
public:         // User declarations
virtual __fastcall TColorForm(TComponent* Owner);
__property String CurrentColor = {read=getColor, write=setColor};
};
```

The *OnClick* event handler for the listbox, *ColorListBoxClick*, sets the value of the *CurrentColor* property each time a new item in the listbox is selected. The event handler gets the string from the listbox containing the color name and assigns it to *CurrentColor*. The *CurrentColor* property uses the setter function, *SetColor*, to store the actual value for the property in the private data member *FColor*:

```
procedure TColorForm.ColorListBoxClick(Sender: TObject);
var
  Index: Integer;
begin
  Index := ColorListBox.ItemIndex;
  if Index >= 0 then
    CurrentColor := ColorListBox.Items[Index]
  else
```

```
    CurrentColor := '';
end;
void __fastcall TColorForm::ColorListBoxClick(TObject *Sender)
{
int index = ColorListBox->ItemIndex;
if (index >= 0) {
CurrentColor = ColorListBox->Items->Strings[index];
}
else
CurrentColor = "";
}
//---------------------------------------------------------------------
void TColorForm::setColor(String s)
{
curColor = s;
}
```

Now suppose that another form within the application, called *ResultsForm*, needs to find out which color is currently selected on *ColorForm* whenever a button (called *UpdateButton*) on *ResultsForm* is clicked. The *OnClick* event handler for *UpdateButton* might look like this:

```
procedure TResultForm.UpdateButtonClick(Sender: TObject);
var
  MainColor: String;
begin
  if Assigned(ColorForm) then
  begin
    MainColor := ColorForm.CurrentColor;
    {do something with the string MainColor}
  end;
end;
void __fastcall TResultsForm::UpdateButtonClick(TObject *Sender)
{
if (ColorForm) {
String s = ColorForm->CurrentColor;
// do something with the color name string
}
}
```

The event handler first verifies that *ColorForm* exists using the *Assigned* function. It then gets the value of *ColorForm's CurrentColor* property.

Alternatively, if *ColorForm* had a public function named *GetColor*, another form could get the current color without using the *CurrentColor* property (for example, `MainColor := ColorForm.GetColor;`). In fact, there's nothing to prevent another form from getting the ColorForm's currently selected color by checking the listbox selection directly:

```
with ColorForm.ColorListBox do
  MainColor := Items[ItemIndex];
String TColorForm::getColor()
{
return curColor;
}
String s = ColorListBox->Items->Strings[ColorListBox->ItemIndex];
```

However, using a property makes the interface to *ColorForm* very straightforward and simple. All a form needs to know about *ColorForm* is to check the value of *CurrentColor*.

See Also

TForm

3.2.4.5.63 Reusing Components and Groups of Components

You can save and reuse work you've done with components using several tools:

- Configure and save groups of components in component templates (see page 1975).
- Save forms, data modules, and projects in the Object Repository (see page 1917). The Repository gives you a central database of reusable elements and lets you use form inheritance to propagate changes.
- Save frames (see page 2012) on the **Tool palette** or in the Repository. Frames use form inheritance and can be embedded into forms or other frames.
- Create a custom component (see page 1313), the most complicated but most flexible way of reusing code. See Overview of Component Creation (see page 1313).

See Also

Developing the Application User Interface: Overview (see page 1983)

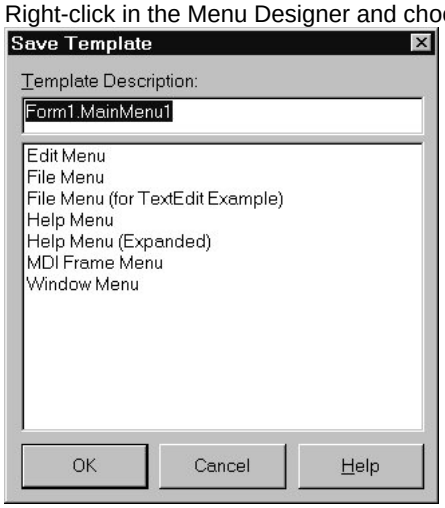
3.2.4.5.64 Saving a Menu as a Template

Any menu you design can be saved as a template so you can use it again. You can use menu templates to provide a consistent look to your applications, or use them as a starting point which you then further customize.

The menu templates you save are stored in your BIN subdirectory as .dmt files.

To save a menu as a template

1. Design the menu you want to be able to reuse. This menu can contain as many items, commands, and submenus as you like; everything in the active Menu Designer window will be saved as one reusable menu.
2. Right-click in the Menu Designer and choose Save As Template. The Save Template dialog box appears.



3. In the Template Description edit box, type a brief description for this menu, and then choose OK. The Save Template dialog box closes, saving your menu design and returning you to the Menu Designer window.

   Note: The description you enter is displayed only in the Save Template, Insert Template, and Delete Templates dialog boxes. It is not related to the *Name* or *Caption* property for the menu.

See Also

TMenu

3.2.4.5.65 Selecting Menu and Toolbar Styles

Just as you can add different colors and icons to individual menus and toolbars, you can select different menu and toolbar styles to give your application a comprehensive look and feel. In addition to the standard style, your application can take on the look of Windows XP, Encarta™, or a custom presentation using a coordinated color scheme. To give your application a coherent look and feel, the IDE uses colormaps.

A colormap can be simple, merely adding the appropriate colors to existing menus and toolbars. Or, a colormap can be complex, altering numerous subtle details of a menu's or toolbar's look and feel, including the smallest button edges or menu shadows. The XP colormap, for example, has numerous subtle refinements for menu and toolbar classes. The IDE handles the details for you, automatically using the appropriate colormaps.

By default, the component library uses the XP style. To centrally select an alternate style for all your application's menus and toolbars, use the *Style* property on the *ActionManager* component.

To select menu and toolbar styles:

1. From the Additional category of the **Tool palette**, drop an *ActionManager* component onto a form.
2. In the **Object Inspector**, select the *Style* property. You can choose from a number of different styles.
3. Once you've selected a style, your application's menus and toolbars will take on the look of the new colormap.

You can customize the look and feel of a style using colormap components.

To customize the look and feel of a colormap:

1. From the Additional category of the **Tool palette**, drop the appropriate colormap component onto a form (for example, *XPColorMap* or *StandardColorMap*). In the **Object Inspector**, you will see numerous properties to adjust appearance, many with drop downs from which you can select alternate values.
2. Change each ToolBar or menu's *ColorMap* property to point to the colormap object that you dropped on the form.
3. In the **Object Inspector**, adjust the colormap's properties to change the appearance of your toolbars and menus as desired.

   Note: Be careful when customizing a colormap. When you select a new, alternate colormap, your old settings will be lost. You may want to save a copy of your application if you want to experiment with alternate settings and possibly return to a previous customization.

See Also

TActionMainMenuBar

TActionToolBar

3.2.4.5.66 Setting Up Action Bands

Because actions do not maintain any "layout" (either appearance or positional) information, Delphi provides action bands which are capable of storing this data. Action bands provide a mechanism that allows you to specify layout information and a set of controls. You can render actions as UI elements such as toolbars and menus.

You organize sets of actions using the Action Manager (TActionManager). You can use standard actions provided or create new actions of your own.

You then create the action bands:

- Use TActionMainMenuBar to create a main menu.
- Use TActionToolBarto create a toolbar.

The action bands act as containers that hold and render sets of actions. You can drag and drop items from the Action Manager editor onto the action band at design time. At runtime, application users can also customize the application's menus or toolbars using a dialog box similar to the Action Manager editor.

See Also

Setting Up Action Lists (see page 2003)

TActionList

TAction

TActionLink

TPopupMenu

3.2.4.5.67 Setting Up Action Lists

Setting up action lists is fairly easy once you understand the basic steps involved:

- Create the action list.
- Add actions to the action list.
- Set properties on the actions.
- Attach clients to the action.

Here are the steps in more detail:

1. Drop a *TActionList* object onto your form or data module. (ActionList is on the Standard category of the **Tool palette**.)
2. Double-click the *TActionList* object to display the Action List editor.

- Use one of the predefined actions listed in the editor: right-click and choose New Standard Action.
- The predefined actions are organized into categories (such as Dataset, Edit, Help, and Window) in the Standard Action Classes dialog box. Select all the standard actions you want to add to the action list and click OK.
- Or, create a new action of your own: right-click and choose New Action.

3. Set the properties of each action in the **Object Inspector**. (The properties you set affect every client of the action.) The Name property identifies the action, and the other properties and events (*Caption*, *Checked*, *Enabled*, *HelpContext*, *Hint*, *ImageIndex*, *ShortCut*, *Visible*, and *Execute*) correspond to the properties and events of its client controls. The client's corresponding properties are typically, but not necessarily, the same name as the corresponding client property. For example, an action's Enabled property corresponds to a *TToolButton's Enabled* property. However, an action's *Checked* property corresponds to a *TToolButton's Down* property.
4. If you use the predefined actions, the action includes a standard response that occurs automatically. If creating your own action, you need to write an event handler that defines how the action responds when fired. See What happens when an action fires (see page 1984) for details.
5. Attach the actions in the action list to the clients that require them:

- Click on the control (such as the button or menu item) on the form or data module. In the **Object Inspector**, the *Action* property lists the available actions.
- Select the one you want.

The standard actions, such as *TEditDelete* or *TDataSetPost*, all perform the action you would expect. You can look at the online reference Help for details on how all of the standard actions work if you need to. If writing your own actions, you'll need to understand more about what happens when the action is fired. See What happens when an action fires (see page 1984) for details.

See Also

Using Action Lists (see page 2007)

3.2.4.5.68 Setting the Appearance of the Cool Bar

The cool bar component offers several useful configuration options. The table below lists some actions you can set to change a tool button's appearance:

Setting a cool button's appearance

To make the cool bar:	Set the toolbar's:
Resize automatically to accommodate the bands it contains	*AutoSize* property to *True.*
Bands maintain a uniform height	*FixedSize* property to *True*.

Reorient to vertical rather than horizontal	*Vertical* property to *True*. This changes the effect of the *FixedSize* property.
Prevent the *Text* properties of the bands from displaying at runtime	*ShowText* property to *False*. Each band in a cool bar has its own *Text* property.
Remove the border around the bar	*BandBorderStyle* to *bsNone.*
Keep users from changing the bands' order at runtime. (The user can still move and resize the bands.)	*FixedOrder* to True.
Create a background image for the cool bar	*Bitmap* property to *TBitmap* object.
Choose a list of images to appear on the left of any band	*Images* property to *TImageList* object.

To assign images to individual bands, select the cool bar and double-click on the *Bands* property in the **Object Inspector**. Then select a band and assign a value to its *ImageIndex* property.

3.2.4.5.69 Setting the Initial Condition of a Speed Button

Speed buttons use their appearance to give the user clues as to their state and purpose. Because they have no caption, it's important that you use the right visual cues to assist users.

The table below lists some actions you can set to change a speed button's appearance:

Setting speed buttons' appearance

To make a speed button:	Set the toolbar's:
Appear pressed	*GroupIndex* property to a value other than zero and its *Down* property to *True*.
Appear disabled	*Enabled* property to *False*.
Have a left margin	*I*ndent property to a value greater than 0.

If your application has a default drawing tool, ensure that its button on the toolbar is pressed when the application starts. To do so, set its *GroupIndex* property to a value other than zero and its *Down* property to *True*.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.70 Setting Tool Button Appearance and Initial Conditions

The table below lists some actions you can set to change a tool button's appearance:

Setting tool buttons' appearance

To make a tool button:	Set the toolbar's:
Appear pressed	(on tool button) *Style* property to *tbsCheck* and *Down* property to *True*.
Appear disabled	*Enabled* property to *False*.
Have a left margin	*I*ndent property to a value greater than 0.

Appear to have "pop-up" borders, thus making the toolbar appear transparent	*Flat* property to *True*.

Note: Using the *Flat* property of *TToolBar* requires version 4.70 or later of COMCTL32.DLL.

To force a new row of controls after a specific tool button, Select the tool button that you want to appear last in the row and set its *Wrap* property to True.

To turn off the auto-wrap feature of the toolbar, set the toolbar's *Wrapable* property to False.

See Also

TToolBar

TCoolBar

TPanel

TToolButton

TSpeedButton

3.2.4.5.71 Sharing Frames

You can share a frame with other developers in two ways:

- Add the frame to the Object Repository.
- Distribute the frame's unit (.pas) and form (.dfm or .xfm) files.

To add a frame to the Repository, open any project that includes the frame, right-click in the Form Designer, and choose Add to Repository. For more information, see Using the Object Repository (see page 1917).

If you send a frame's unit and form files to other developers, they can open them and add them to the **Tool palette**. If the frame has other frames embedded in it, they will have to open it as part of a project.

See Also

Working with Frames (see page 2012)

Using and Modifying Frames (see page 2008)

Creating Frames (see page 1977)

3.2.4.5.72 Specifying Accelerator Keys and Keyboard Shortcuts

Accelerator keys enable the user to access a menu command from the keyboard by pressing `Alt`+ the appropriate letter, indicated in your code by the preceding ampersand. The letter after the ampersand appears underlined in the menu.

Delphi automatically checks for duplicate accelerators and adjusts them at runtime. This ensures that menus built dynamically at runtime contain no duplicate accelerators and that all menu items have an accelerator. You can turn off this automatic checking by setting the *AutoHotkeys* property of a menu item to *maManual***.**

To specify an accelerator, add an ampersand in front of the appropriate letter. For example, to add a Save menu command with the S as an accelerator key, type `&Save`.

Keyboard shortcuts enable the user to perform the action without using the menu directly, by typing in the shortcut key combination.

To specify a keyboard shortcut, use the **Object Inspector** to enter a value for the ShortCut property, or select a key combination from the drop-down list. This list is only a subset of the valid combinations you can type in.

When you add a shortcut, it appears next to the menu item caption.

Warning: Keyboard shortcuts, unlike accelerator keys, are not checked automatically for duplicates. You must ensure uniqueness yourself.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.73 Specifying the Active Menu: Menu Property

The *Menu* property specifies the active menu for the form. Menu-merging operations apply only to the active menu. If the form contains more than one menu component, you can change the active menu at runtime by setting the *Menu* property in code. For example,

```
Form1.Menu := SecondMenu;
Form1->Menu = SecondMenu;
```

See Also

TMenu

3.2.4.5.74 Switching Between Menus at Design Time

If you're designing several menus for your form, you can use the Menu Designer context menu or the **Object Inspector** to easily select and move among them.

To use the context menu to switch between menus in a form:

1. Right-click in the Menu Designer and choose Select Menu. The Select Menu dialog box appears.



This dialog box lists all the menus associated with the form whose menu is currently open in the Menu Designer.

2. From the list in the Select Menu dialog box, choose the menu you want to view or edit.

To use the Object Inspector to switch between menus in a form:

1. Give focus to the form whose menus you want to choose from.
2. From the Component list, select the menu you want to edit.
3. On the Properties page of the **Object Inspector**, select the *Items* property for this menu, and then either click the ellipsis button, or double-click [Menu].

See Also

TMenu

3.2.4.5.75 Controlling Application Behavior

TApplication, TScreen, and TForm are the classes that form the backbone of all applications by controlling the behavior of your project. The *TApplication* class forms the foundation of an application by providing properties and methods that encapsulate the behavior of a standard program. *TScreen* is used at runtime to keep track of forms and data modules that have been loaded as well as maintaining system-specific information such as screen resolution and available display fonts. Instances of the *TForm* class are the building blocks of your application's user interface. The windows and dialog boxes in your application are based on *TForm*.

See Also

TApplication

TScreen

3.2.4.5.76 Updating Actions

When the application is idle, the *OnUpdate* event occurs for every action that is linked to a control or menu item that is showing. This provides an opportunity for applications to execute centralized code for enabling and disabling, checking and unchecking, and so on. For example, the following code illustrates the *OnUpdate* event handler for an action that is "checked" when the toolbar is visible:

```
procedure TForm1.Action1Update(Sender: TObject);
begin
{ Indicate whether ToolBar1 is currently visible }
(Sender as TAction).Checked := ToolBar1.Visible;
end;
void __fastcall TForm1::Action1Update(TObject *Sender)
{
// Indicate whether ToolBar1 is currently visible
((TAction *)Sender)->Checked = ToolBar1->Visible;
}
```

Warning: Do not add time-intensive code to the *OnUpdate* event handler. This executes whenever the application is idle. If the event handler takes too much time, it will adversely affect performance of the entire application.

See Also

TActionList

TAction

TActionLink

3.2.4.5.77 Using Action Lists

Note: The contents of this topic apply to setting up toolbars and menus for cross-platform development. For Windows development you can also use the methods described here. However, using action bands instead is simpler and offers more options. The action lists will be handled automatically by the Action Manager. See Organizing actions for toolbars and menus (see page 1992) for details.

Action lists maintain a list of actions that your application can take in response to something a user does. By using action objects, you centralize the functions performed by your application from the user interface. This lets you share common code for performing actions (for example, when a toolbar button and menu item do the same thing), as well as providing a single, centralized way to enable and disable actions depending on the state of your application.

See Also

TActionList

TAction

TActionLink

Setting Up Action Lists (see page 2003)

3.2.4.5.78 Using and Modifying Frames

To use a frame in an application, you must place it, directly or indirectly, on a form. You can add frames directly to forms, to other frames, or to other container objects such as panels and scroll boxes.

The Form Designer provides two ways to add a frame to an application:

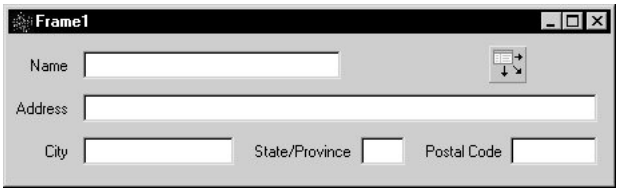
- Select a frame from the **Tool palette** and drop it onto a form, another frame, or another container object. If necessary, the Form Designer asks for permission to include the frame's unit file in your project.
- Select *Frames* from the Standard category of the **Tool palette** and click on a form or another frame. A dialog appears with a list of frames that are already included in your project; select one and click OK.

When you drop a frame onto a form or other container, Delphi declares a new class that descends from the frame you selected. (Similarly, when you add a new form to a project, Delphi declares a new class that descends from *TForm*.) This means that changes made later to the original (ancestor) frame propagate to the embedded frame, but changes to the embedded frame do not propagate backward to the ancestor.

Suppose, for example, that you wanted to assemble a group of data-access components and data-aware controls for repeated use, perhaps in more than one application. One way to accomplish this would be to collect the components into a component template; but if you started to use the template and later changed your mind about the arrangement of the controls, you would have to go back and manually alter each project where the template was placed.

If, on the other hand, you put your database components into a frame, later changes would need to be made in only one place; changes to an original frame automatically propagate to its embedded descendants when your projects are recompiled. At the same time, you are free to modify any embedded frame without affecting the original frame or other embedded descendants of it. The only limitation on modifying embedded frames is that you cannot add components to them.

A frame with data-aware controls and a data source component:



In addition to simplifying maintenance, frames can help you to use resources more efficiently. For example, to use a bitmap or other graphic in an application, you might load the graphic into the Picture property of a *TImage* control. If, however, you use the same graphic repeatedly in one application, each *Image* object you place on a form will result in another copy of the graphic being added to the form's resource file. (This is true even if you set *TImage.Picture* once and save the *Image* control as a component template.) A better solution is to drop the *Image* object onto a frame, load your graphic into it, then use the frame where you want the graphic to appear. This results in smaller form files and has the added advantage of letting you change the graphic everywhere it occurs simply by modifying the *Image* on the original frame.

See Also

Working with Frames (see page 2012)

Creating Frames (see page 1977)

Sharing Frames (see page 2005)

3.2.4.5.79 Using Menu Templates

Several predesigned menus, or menu templates, contain frequently used commands. You can use these menus in your applications without modifying them (except to write code), or you can use them as a starting point, customizing them as you would a menu you originally designed yourself. Menu templates do not contain any event handler code.

The menu templates are stored in the BIN subdirectory in a default installation and have a .dmt extension.

You can also save as a template any menu that you design using the Menu Designer. After saving a menu as a template, you can use it as you would any predesigned menu. If you decide you no longer want a particular menu template, you can delete it from the list.

To add a menu template to your application

1. Right-click the Menu Designer and choose Insert From Template. (If there are no templates, the Insert From Template option appears dimmed in the context menu.) The Insert Template dialog box opens, displaying a list of available menu templates.



2. Select the menu template you want to insert, then press Enter or choose OK. This inserts the menu into your form at the cursor's location. For example, if your cursor is on a menu item in a list, the menu template is inserted above the selected item. If your cursor is on the menu bar, the menu template is inserted to the left of the cursor.

To delete a menu template

1. Right-click the Menu Designer and choose Delete Templates. (If there are no templates, the Delete Templates option appears dimmed in the context menu.) The Delete Templates dialog box opens, displaying a list of available templates.
2. Select the menu template you want to delete, and press `Del`. Delphi deletes the template from the templates list and from your hard disk.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.80 Using the Main Form

The first form you create and save in a project becomes, by default, the project's main form, which is the first form created at runtime. As you add forms to your projects, you might decide to designate a different form as your application's main form. Also, specifying a form as the main form is an easy way to test it at runtime, because unless you change the form creation order, the main form is the first form displayed in the running application.

To change the project main form:

1. Choose **Project** ▸ **Options** and select the Forms page.

2. In the Main Form combo box, select the form you want to use as the project's main form and choose OK.

Now if you run the application, the form you selected as the main form is displayed.

See Also

TForm

Hiding the Main Form (see page 1986)

3.2.4.5.81 Using the Menu Designer Context Menu

The Menu Designer context menu provides quick access to the most common Menu Designer commands, and to the menu template options. (For more information about menu templates, refer to Using menu templates (see page 2009).)

To display the context menu, right-click the Menu Designer window, or press *Alt+F10* when the cursor is in the Menu Designer window.

Commands on the context menu

The following table summarizes the commands on the Menu Designer context menu.

Menu Designer context menu commands

Menu command	Action
Insert	Inserts a placeholder above or to the left of the cursor.
Delete	Deletes the selected menu item (and all its sub-items, if any).
Create Submenu	Creates a placeholder at a nested level and adds an arrow to the right of the selected menu item.
Select Menu	Opens a list of menus in the current form. Double-clicking a menu name opens the designer window for the menu.
Save As Template	Opens the Save Template dialog box, where you can save a menu for future reuse.
Insert From Template	Opens the Insert Template dialog box, where you can select a template to reuse.
Delete Templates	Opens the Delete Templates dialog box, where you can choose to delete any existing templates.
Insert From Resource	Opens the Insert Menu from Resource file dialog box, where you can choose a .rc or .mnu file to open in the current form.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.82 Creating a Form Instance Using a Local Variable

A safer way to create a unique instance of a *modal form* is to use a local variable in the event handler as a reference to a new instance. If a local variable is used, it does not matter whether *ResultsForm* is auto-created or not. The code in the event handler makes no reference to the global form variable. For example:

```
procedure TMainForm.Button1Click(Sender: TObject);
var
  RF:TResultForm;
begin
  RF:=TResultForm.Create(self)
  RF.ShowModal;
  RF.Free;
end;
```

```
void __fastcall TMainMForm::FirstButtonClick(TObject *Sender)
{
TResultsForm *rf = new TResultsForm(this);
rf->ShowModal();
delete rf;
}
```

Notice how the global instance of the form is never used in this version of the event handler.

Typically, applications use the global instances of forms. However, if you need a new instance of a modal form, and you use that form in a limited, discrete section of the application, such as a single function, a local instance is usually the safest and most efficient way of working with the form.

Of course, you cannot use local variables in event handlers for modeless forms because they must have global scope to ensure that the forms exist for as long as the form is in use. *Show* returns as soon as the form opens, so if you used a local variable, the local variable would go out of scope immediately.

See Also

TForm

3.2.4.5.83 Viewing the Menu

You can view your menu in the form at design time without first running your program code. (Pop-up menu components are visible in the form at design time, but the pop-up menus themselves are not. Use the Menu Designer to view a pop-up menu at design time.)

To view the menu:

1. If the form is visible, click the form, or from the View menu, choose the form whose menu you want to view.
2. If the form has more than one menu, select the menu you want to view from the form's Menu property drop-down list. The menu appears in the form exactly as it will when you run the program.

See Also

TMenu

Creating and Managing Menus (see page 1975)

3.2.4.5.84 What Is an Action?

As you are developing your application, you can create a set of actions that you can use on various UI elements. You can organize them into categories that can be dropped onto a menu as a set (for example, Cut, Copy, and Paste) or one at a time (for example, **Tools** ▸ **Customize**).

An action corresponds to one or more elements of the user interface, such as menu commands or toolbar buttons. Actions serve two functions: (1) they represent properties common to the user interface elements, such as whether a control is enabled or checked, and (2) they respond when a control fires, for example, when the application user clicks a button or chooses a menu item. You can create a repertoire of actions that are available to your application through menus, through buttons, through toolbars, context menus, and so on.

Actions are associated with other components:

- **Clients:** One or more clients use the action. The client most often represents a menu item or a button (for example, *TToolButton*, *TSpeedButton*, *TMenuItem*, *TButton*, *TCheckBox, TRadioButton*, and so on). Actions also reside on ActionBand components such as TActionMainMenuBar *and* TActionToolBar. When the client receives a user command (such as a mouse click), it initiates an associated action. Typically, a client's *OnClick* event is associated with its action's *OnExecute* event.
- **Target:** The action acts on the target. The target is usually a control, such as a memo or a data control. Component writers can create actions specific to the needs of the controls they design and use, and then package those units to create more

modular applications. Not all actions use a target. For example, the standard help actions ignore the target and simply launch the help system.

A target can also be a component. For example, data controls change the target to an associated dataset.

The client influences the action—the action responds when a client fires the action. The action also influences the client—action properties dynamically update the client properties. For example, if at runtime an action is disabled (by setting its *Enabled* property to *False*), every client of that action is disabled, appearing grayed.

You can add, delete, and rearrange actions using the Action Manager or the Action List editor (displayed by double-clicking an action list object, *TActionList*). These actions are later connected to client controls. See Creating toolbars and menus (see page 1978) and, for cross-platform development, Setting up action lists (see page 2003) for details.

See Also

TActionManager

TActionList

TAction

TActionLink

Setting Up Action Lists (see page 2003)

3.2.4.5.85 Working at the Application Level

The global variable *Application*, of type *TApplication*, is in every VCL-based application. *Application* encapsulates your application as well as providing many functions that occur in the background of the program. For instance, *Application* handles how you call a Help file from the menu of your program. Understanding how *TApplication* works is more important to a component writer than to developers of stand-alone applications, but you should set the options that *Application* handles in the **Project ▶ Options** Application page when you create a project.

In addition, *Application* receives many events that apply to the application as a whole. For example, the *OnActivate* event lets you perform actions when the application first starts up, the *OnIdle* event lets you perform background processes when the application is not busy, the *OnMessage* event lets you intercept Windows messages (on Windows only), the *OnEvent* event lets you intercept events, and so on. Although you can't use the IDE to examine the properties and events of the global *Application* variable, another component, *TApplicationEvents*, intercepts the events and lets you supply event-handlers using the IDE.

See Also

TApplication

3.2.4.5.86 Working with Frames

A frame (*TFrame*), like a form, is a container for other components. It uses the same ownership mechanism as forms for automatic instantiation and destruction of the components on it, and the same parent-child relationships for synchronization of component properties.

In some ways, however, a frame is more like a customized component than a form. Frames can be saved on the **Tool palette** for easy reuse, and they can be nested within forms, other frames, or other container objects. After a frame is created and saved, it continues to function as a unit and to inherit changes from the components (including other frames) it contains. When a frame is embedded in another frame or form, it continues to inherit changes made to the frame from which it derives.

Frames are useful to organize groups of controls that are used in multiple places in your application. For example, if you have a bitmap that is used on multiple forms, you can put it in a frame and only one copy of that bitmap is included in the resources of your application. You could also describe a set of edit fields that are intended to edit a table with a frame and use that whenever you want to enter data into the table.

Creating frames (see page 1977)

Using and modifying frames (see page 2008)

Sharing frames (see page 2005)

See Also

Controlling Application Behavior (see page 2007)

Overview of Component Creation (see page 1313)

Creating and Using Component Templates (see page 1975)

Reusing Components and Groups of Components (see page 2000)

3.2.4.5.87 Writing Action Components

You can also create your own predefined action classes. When you write your own action classes, you can build in the ability to execute on certain target classes of objects. Then, you can use your custom actions in the same way you use predefined action classes. That is, when the action can recognize and apply itself to a target class, you can simply assign the action to a client control, and it acts on the target with no need to write an event handler.

Component writers can use the classes in the QStdActns and DBActns units as examples for deriving their own action classes to implement behaviors specific to certain controls or components. The base classes for these specialized actions (*TEditAction, TWindowAction*, and so on) generally override *HandlesTarget*, *UpdateTarget*, and other methods to limit the target for the action to a specific class of objects. The descendant classes typically override *ExecuteTarget* to perform a specialized task. These methods are described here:

Methods overriden by base classes of specific actions

Method	Description
HandlesTarget	Called automatically when the user invokes an object (such as a tool button or menu item) that is linked to the action. The *HandlesTarget* method lets the action object indicate whether it is appropriate to execute at this time with the object specified by the *Target* parameter as a "target". See How actions find their targets (see page 1987) for details.
UpdateTarget	Called automatically when the application is idle so that actions can update themselves according to current conditions. Use in place of *OnUpdateAction*. See Updating actions (see page 2007) for details.
ExecuteTarget	Called automatically when the action fires in response to a user action in place of *OnExecute* (for example, when the user selects a menu item or presses a tool button that is linked to this action). See What happens when an action fires (see page 1984) for details.

When you write your own action classes, it is important to understand the following:

- How actions find their targets (see page 1987)
- Registering actions (see page 1996)

See Also

TActionList

TAction

TActionLink

3.2.4.5.88 Using Windows Common Dialog Boxes

One of the commonly used dialog box components is *TOpenDialog*. This component is usually invoked by a New or Open menu item under the File option on the main menu bar of a form. The dialog box contains controls that let you select groups of files

using a wildcard character and navigate through directories.

The *TOpenDialog* component makes an Open dialog box available to your application. The purpose of this dialog box is to let a user specify a file to open. You use the *Execute* method to display the dialog box.

When the user chooses OK in the dialog box, the user's file is stored in the *TOpenDialog FileName* property, which you can then process as you want.

The following code can be placed in an *Action* and linked to the *Action* property of a *TMainMenu* subitem or be placed in the subitem's *OnClick* event:

```
if OpenDialog1.Execute then
   filename := OpenDialog1.FileName;
if(OpenDialog1->Execute()){
   filename = OpenDialog1->FileName;
};
```

This code will show the dialog box and if the user presses the OK button, it will copy the name of the file into a previously declared *AnsiString* variable named `filename`.

3.2.4.6 Exception handling

Topics

Name	Description
Unwinding Exceptions (C++) (see page 2016)	When an exception is thrown, the runtime library takes the thrown object, gets the type of the object, and looks upward in the call stack for a handler whose type matches the type of the thrown object. Once a handler is found, the RTL unwinds the stack to the point of the handler, and executes the handler. In the unwind process, the RTL calls destructors for all local objects in the stack frames between where the exception was thrown and where it is caught. If a destructor causes an exception to be raised during stack unwinding and does not handle... more (see page 2016)
Exception Handling Options (C++) (see page 2016)	The following exception handling options are available for bcc32:
Constructors In Exception Handling (C++) (see page 2017)	Class constructors can throw exceptions if they cannot successfully construct an object. If a constructor throws an exception, that object's destructor is not necessarily called. Destructors are called only for the base classes and for those objects that were fully constructed inside the classes since entering the try block. **Note:** This does not apply to VCL base classes.
Defining Protected Blocks (see page 2017)	To prepare for exceptions, you place statements that might raise them in a try block (see page 2026). If one of these statements does raise an exception, control is transferred to an exception handler that handles that type of exception, then leaves the block. The exception handler is said to *catch* the exception and specifies the actions to take. By using try blocks and exception handlers, you can move error checking and error handling out of the main flow of your algorithms, resulting in simpler, more readable code. You start a protected block using the keyword **try**. The exception handler must... more (see page 2017)
Exception Handling (see page 2018)	Exceptions are exceptional conditions that require special handling. They include errors that occur at runtime, such as divide by zero, and the exhaustion of free store. Exception handling provides a standard way of dealing with errors, discovering both anticipated and unanticipated problems, and enables developers to recognize, track down, and fix bugs. When an error occurs, the program raises an exception, meaning it creates an exception object and rolls back the stack to the first point it finds where you have code to handle the exception. The exception object usually contains information about what happened. This allows another part of... more (see page 2018)
Exception-handling Statements (see page 2018)	The exception handling block starts with the **except** keyword and ends with the keyword **end**. These two keywords are actually part of the same statement as the try block. That is, both the try block and the exception handling block are considered part of a single **try...except** statement. Inside the exception handling block, you include one or more exception handlers. An exception handler is a statement of the form
Throwing An Exception (C++) (see page 2019)	To raise an exception in C++, use the **throw** keyword. Objects in C++ can be thrown by value, or pointer:

Raising an Exception (see page 2020)	To indicate a disruptive error condition, you can raise an exception by constructing an instance of an exception object that describes the error condition and calling the reserved word **raise**. To raise an exception, call the reserved word **raise**, followed by an instance of an exception object. This establishes the exception as coming from a particular address. When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself. For example, given the following declaration,
Reraising Exceptions (see page 2020)	Sometimes when you handle an exception locally, you want to augment the handling in the enclosing block, rather than replace it. Of course, when your local handler finishes its handling, it destroys the exception instance, so the enclosing block's handler never gets to act. You can, however, prevent the handler from destroying the exception, giving the enclosing handler a chance to respond. You do this by using the raise command with no arguments. This is called reraising or rethrowing the exception. The following example illustrates this technique:
Smart Pointers (C++) (see page 2021)	If you have local variables that are pointers to objects and an exception is thrown, these pointers are not automatically deleted. This is because there is no good way for the compiler to distinguish between a pointer to data that was allocated for this function only and any other pointer. The class that you can use to ensure that objects allocated for local use are destroyed in the even of an exception is `auto_ptr`. There is a special case in which memory is freed for a pointer allocated in a function:
Scope of Exception Handlers (see page 2021)	You do not need to provide handlers for every kind of exception in every block. You only need handlers for exceptions that you want to handle specially within a particular block. If a block does not handle a particular exception, execution leaves that block and returns to the block that contains it (or returns to the code that called the block), with the exception still raised. This process repeats with increasingly broad scope until either execution reaches the outermost scope of the application or a block at some level handles the exception. Thus, you can nest your exception handling code.... more (see page 2021)
Silent Exceptions (see page 2022)	VCL applications handle most exceptions that your code doesn't specifically handle by displaying a message box that shows the message string from the exception object. You can also define "silent" exceptions that do not, by default, cause the application to show the error message. Silent exceptions are useful when you don't intend to report an exception to the user, but you want to abort an operation. Aborting an operation is similar to using the *Break* or *Exit* procedures to break out of a block, but can break out of several nested levels of blocks. Silent exceptions all descend from the... more (see page 2022)
Writing a finally Block (C++) (see page 2023)	`bcc32` includes extensions to the C++ language that let it use finally blocks as well. Like exception handlers, a finally block must appear directly after the **try** block, but it is introduced by the__**finally** keyword instead of **catch**.
Writing a Finally Block (Delphi) (see page 2024)	Finally blocks are introduced by the keyword **finally**. They are part of a **try..finally** statement, which has the following form:
Writing Exception Handlers (see page 2024)	The exception handling block appears immediately after the try block. This block incudes one or more exception handlers. An exception handler provides a specific response to a specific kind of exception. Handling an exception clears the error condition and destroys the exception object, which allows the application to continue execution. You typically define exception handlers to allow your applications to recover from errors and continue running. Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as "File not found," are... more (see page 2024)
Writing finally Blocks (see page 2025)	An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code. However, there are times when you do not need to handle the exception, but you do have code that you want to execute after the protected block, even if an exception occurs. Typically, such code handles cleanup issues, such as freeing resources that were allocated before the protected block. By using finally blocks, you can ensure that if your application allocates resources, it also releases them, even if an exception occurs. Thus, if your application allocates memory, you can make... more (see page 2025)
Writing the Try Block (see page 2026)	The first part of a protected block is the try block. The try block contains code that can potentially raise an exception. The exception can be raised either directly in the try block, or by code that is called by statements in the try block. That is, if code in a try block calls a routine that doesn't define its own exception handler, then any exceptions raised inside that routine cause execution to pass to the exception-handler associated with the try block. Keep in mind that exceptions don't come just from your code. A call to an RTL routine or... more (see page 2026)

Handling Classes of Exceptions (see page 2027)	Exceptions are always represented by classes. As such, you usually work with a hierarchy of exception classes. For example, VCL defines the *ERangeError* exception as a descendant of *EIntError*. When you provide an exception handler for a base exception class, it catches not only direct instances of that class, but instances of any of its descendants as well. For example, the following exception handler handles all integer math exceptions, including *ERangeError*, *EDivByZero*, and *EIntOverflow*:
VCL Exception Classes (see page 2028)	VCL includes a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions. All VCL exception classes descend from one root object called Exception. *Exception* provides a consistent interface for applications to handle exceptions. It provides the string for the message that VCL exceptions display by default. The following table lists a selection of the exception classes defined in VCL: ***Selected exception classes***
Default Exception Handling in VCL (see page 2029)	If your application code does not catch and handle the exceptions that are raised, the exceptions are ultimately caught and handled by the *HandleException* method of the global *Application* object. For all exceptions but *EAbort*, *HandleException* calls the *OnException* event handler, if one exists. If there is no *OnException* event handler (and the exception is not *EAbort*), *HandleException* displays a message box with the error message associated with the exception. There are certain circumstances where *HandleException* does not get called. Exceptions that occur before or after the execution of the application's Run method are not caught and handled... more (see page 2029)
Defining Your Own VCL Exceptions (see page 2029)	Because VCL exceptions are classes, defining a new kind of exception is as simple as declaring a new class type. Although you can raise any object instance as an exception, the standard VCL exception handlers handle only exceptions that descend from *Exception*. New exception classes should be derived from *Exception* or one of the other standard exceptions. That way, if you raise your new exception in a block of code that isn't protected by an exception handler specific to that exception, one of the standard handlers will handle it instead. For example, consider the following declaration:
Handling Exceptions in VCL Applications (see page 2030)	If you use VCL components or the VCL runtime library in your applications, you need to understand the VCL exception handling mechanism. Exceptions are built into many VCL classes and routines and they are thrown automatically when something unexpected occurs. Typically, these exceptions indicate programming errors that would otherwise generate a runtime error. A limited number of these classes is described in VCL Exception Classes (see page 2028). The mechanics of handling component exceptions are no different than handling any other type of exception. If you do not handle the exception, VCL handles it in a default manner (see page 2029). Typically, a message... more (see page 2030)

3.2.4.6.1 Unwinding Exceptions (C++)

When an exception is thrown, the runtime library takes the thrown object, gets the type of the object, and looks upward in the call stack for a handler whose type matches the type of the thrown object. Once a handler is found, the RTL unwinds the stack to the point of the handler, and executes the handler.

In the unwind process, the RTL calls destructors for all local objects in the stack frames between where the exception was thrown and where it is caught. If a destructor causes an exception to be raised during stack unwinding and does not handle it, terminate is called. Destructors are called by default, but you can switch off the default by using the -xd compiler option.

Note: During the unwind process, the RTL does not call destructors for objects that are allocated on the heap rather than the stack. This is why, for example, VCL applications use finally

blocks to ensure that VCL objects, which are always allocated on the heap, are properly freed. There is one exception to this rule, which is the use of safe pointers.

3.2.4.6.2 Exception Handling Options (C++)

The following exception handling options are available for bcc32:

-x	Enables C++ exception handling. Enabled by default.
-xd	Enables destructor cleanup. Calls destructors for all automatically declared objects between the scope of the **catch** and **throw** statements when an exception is thrown. Enabled by default.
-xp	Enables the program to use the `__ThrowFileName` global to obtain the file where the exception occurred and the `__ThrowLineNumber` global to access the line number from where the C++ exception was thrown. Disabled by default.

3.2.4.6.3 Constructors In Exception Handling (C++)

Class constructors can throw exceptions if they cannot successfully construct an object. If a constructor throws an exception, that object's destructor is not necessarily called. Destructors are called only for the base classes and for those objects that were fully constructed inside the classes since entering the try block.

Note: This does not apply to VCL base classes.

See Also

Exception Handling Options (see page 2016)

Unwinding Exceptions (C++) (see page 2016)

3.2.4.6.4 Defining Protected Blocks

To prepare for exceptions, you place statements that might raise them in a try block (see page 2026). If one of these statements does raise an exception, control is transferred to an exception handler that handles that type of exception, then leaves the block. The exception handler is said to *catch* the exception and specifies the actions to take. By using try blocks and exception handlers, you can move error checking and error handling out of the main flow of your algorithms, resulting in simpler, more readable code.

You start a protected block using the keyword **try**. The exception handler must immediately follow the try block. It is introduced by the keyword **except**, and signals the end of the try block. This syntax is illustrated in the following code. If the *SetFieldValue* method fails and raises an *EIntegerRange* exception, execution jumps to the exception-handling part, which displays an error message. Execution resumes outside the block.

```
try
  SetFieldValue(dataField, userValue);
except
  on E: EIntegerRange do
    ShowMessage(Format('Expected value between %d and %d, but got %d',
                       E.Min, E.Max, E.Value));
end;
  .  { execution resumes here, outside the protected block }
  .
  .
try
{
SetFieldValue(dataField, userValue);
}
catch (EIntegerRange &E)
{
  ShowMessage(Format("Expected value between %d and %d, but got %d\n",
              E.Min, E.Max, E.Value));
}
// execution resumes here, outside the protected block
```

You must have an *exception handling block* (described in Writing Exception Handlers (see page 2024)) or a *finally block*

(described in Writing Finally Blocks (see page 2025)) immediately after the try block. An exception handling block should include a handler for each exception that the statements in the try block can generate.

See Also

Writing Exception Handlers (see page 2024)

3.2.4.6.5 Exception Handling

Exceptions are exceptional conditions that require special handling. They include errors that occur at runtime, such as divide by zero, and the exhaustion of free store. Exception handling provides a standard way of dealing with errors, discovering both anticipated and unanticipated problems, and enables developers to recognize, track down, and fix bugs.

When an error occurs, the program raises an exception, meaning it creates an exception object and rolls back the stack to the first point it finds where you have code to handle the exception. The exception object usually contains information about what happened. This allows another part of the program to diagnose the cause of the exception.

To make your applications robust, your code needs to recognize exceptions when they occur and respond to them. If you don't specify a response, the application presents a message box describing the error. Your job, then, is to recognize places where errors might happen, and define responses, particularly in areas where errors could cause the loss of data or system resources.

When you create a response to an exception, you do so on blocks of code. When you have a series of statements that all require the same kind of response to errors, you can group them into a block and define error responses that apply to the whole block.

Blocks with specific responses to exceptions are called protected blocks because they can guard against errors that might otherwise either terminate the application or damage data.

See Defining Protected Blocks (see page 2017) for details on how to create and handle exceptions.

For information on using exceptions with the routines and classes in VCL, see Handling Exceptions in VCL Applications (see page 2030).

See Also

Creating Applications (see page 1883)

3.2.4.6.6 Exception-handling Statements

The exception handling block starts with the **except** keyword and ends with the keyword **end**. These two keywords are actually part of the same statement as the try block. That is, both the try block and the exception handling block are considered part of a single **try...except** statement.

Inside the exception handling block, you include one or more exception handlers. An exception handler is a statement of the form

```
on <type of exception> do <statement>;
```

For example, the following exception handling block includes multiple exception handlers for different exceptions that can arise from an arithmetic computation:

```
try
{ calculation statements }
except
  on EZeroDivide do Value := MAXINT;
  on EIntOverflow do Value := 0;
  on EIntUnderflow do Value := 0;
end;
```

Much of the time, as in the previous example, the exception handler doesn't need any information about an exception other than its type, so the statements following **on..do** are specific only to the type of exception. In some cases, however, you might need

some of the information contained in the exception instance.

To read specific information about an exception instance in an exception handler, you use a special variation of **on..do** that gives you access to the exception instance. The special form requires that you provide a temporary variable to hold the instance. For example:

```
on E: EIntegerRange do
  ShowMessage(Format('Expected value between %d and %d', E.Min, E.Max));
```

The temporary variable (E in this example) is of the type specified after the colon (*EIntegerRange* in this example). You can use the as operator to typecast the exception into a more specific type if needed.

Warning: Never destroy the temporary exception object. Handling an exception automatically destroys the exception object. If you destroy the object yourself, the application attempts to destroy the object again, generating an access violation.

You can provide a single default exception handler to handle any exceptions for which you haven't provided specific handlers. To do that, add an else part to the exception-handling block:

```
try
{ statements }
except
  on ESomething do
    { specific exception-handling code };
  else
    { default exception-handling code };
end;
```

Adding default exception handling to a block guarantees that the block handles every exception in some way, thereby overriding all handling from any containing block.

Warning: It is not advisable to use this all-encompassing default exception handler. The else clause handles all exceptions, including those you know nothing about. In general, your code should handle only exceptions you actually know how to handle. If you want to handle cleanup and leave the exception handling to code that has more information about the exception and how to handle it, then you can do so using a finally block. For details about finally blocks, see Writing Finally Blocks (see page 2025).

See Also

Raising an Exception (see page 2020)

Writing a Finally Block (see page 2024)

Handling Classes of Exceptions (see page 2027)

Scope of Exception Handlers (see page 2021)

Reraising Exceptions (see page 2020)

3.2.4.6.7 Throwing An Exception (C++)

To raise an exception in C++, use the **throw** keyword. Objects in C++ can be thrown by value, or pointer:

```
// throw an object, to be caught by value or reference
throw EIntegerRange(0, 10, userValue);
// throw an object to be caught by pointer
throw new EIntegerRange(0, 10, userValue);
```

Tip: Throw exceptions by value and catch exceptions by reference to prevent memory leaks. If you catch an exception by pointer, you may not be able to delete the exception object.

Note: To throw an exception by value, it must have a public copy constructor and public destructor.

In addition, the throw statement can throw other types as well. Although it is not recommended, C++ lets you throw primitive types, such as integers or pointers:

```
throw 1;       // throw an int
throw "catastrophic error";           // throw a char *
```

In most cases, you want to throw exception objects because they can provide a more complete description of an error.

3.2.4.6.8 Raising an Exception

To indicate a disruptive error condition, you can raise an exception by constructing an instance of an exception object that describes the error condition and calling the reserved word **raise**.

To raise an exception, call the reserved word **raise**, followed by an instance of an exception object. This establishes the exception as coming from a particular address. When an exception handler actually handles the exception, it finishes by destroying the exception instance, so you never need to do that yourself.

For example, given the following declaration,

```
type
  EPasswordInvalid = class(Exception);
```

you can raise a "password invalid" exception at any time by calling raise with an instance of *EPasswordInvalid*, like this:

```
if Password <> CorrectPassword then
  raise EPasswordInvalid.Create('Incorrect password entered');
```

Raising an exception sets the *ErrorAddr* variable in the System unit to the address where the application raised the exception. You can refer to *ErrorAddr* in your exception handlers, for example, to notify the user where the error occurred. You can also specify a value in the raise clause that appears in *ErrorAddr* when an exception occurs.

Warning: Do not assign a value to *ErrorAddr* yourself. It is intended as read-only.

To specify an error address for an exception, add the reserved word *at* after the exception instance, followed by an address expression such as an identifier.

See Also

Exception-handling Statements (see page 2018)

Writing a Finally Block (see page 2024)

Reraising Exceptions (see page 2020)

3.2.4.6.9 Reraising Exceptions

Sometimes when you handle an exception locally, you want to augment the handling in the enclosing block, rather than replace it. Of course, when your local handler finishes its handling, it destroys the exception instance, so the enclosing block's handler never gets to act. You can, however, prevent the handler from destroying the exception, giving the enclosing handler a chance to respond. You do this by using the raise command with no arguments. This is called reraising or rethrowing the exception. The following example illustrates this technique:

```
try
{ statements }
  try
{ special statements }
  except
    on ESomething do
    begin
{ handling for only the special statements }
      raise;{ reraise the exception }
    end;
  end;
except
  on ESomething do ...;{ handling you want in all cases }
```

```
end;
try
{
  // statements
  try
  {
    // special statements
  }
  catch (const ESomething &E)
  {
    // handling for only the special statements;
  }
}
catch (const ESomething &E)
{
  // handling you want in all cases
}
```

If code in the *statements* part raises an *ESomething* exception, only the handler in the outer exception-handling block executes. However, if code in the *special statements* part raises *ESomething*, the handling in the inner exception-handling block executes, followed by the more general handling in the outer exception-handling block. By reraising exceptions, you can easily provide special handling for exceptions in special cases without losing (or duplicating) the existing handlers.

If the handler wants to throw a different exception, it can use the raise or throw statement in the normal way, as described in Raising an Exception (see page 2020).

See Also

Exception-handling Statements (see page 2018)

Handling Classes of Exceptions (see page 2027)

Scope of Exception Handlers (see page 2021)

3.2.4.6.10 Smart Pointers (C++)

If you have local variables that are pointers to objects and an exception is thrown, these pointers are not automatically deleted. This is because there is no good way for the compiler to distinguish between a pointer to data that was allocated for this function only and any other pointer. The class that you can use to ensure that objects allocated for local use are destroyed in the even of an exception is `auto_ptr`. There is a special case in which memory is freed for a pointer allocated in a function:

```
auto_ptr< TMyObject > pMyObject( new TMyObject );
```

In this example, if the constructor for `TMyObject` throws an exception, then the pointer to the object allocated for TMYObject will be deleted by the RTL when it unwinds the exception. This is the only time that the compiler automatically deletes a pointer value for you.

3.2.4.6.11 Scope of Exception Handlers

You do not need to provide handlers for every kind of exception in every block. You only need handlers for exceptions that you want to handle specially within a particular block.

If a block does not handle a particular exception, execution leaves that block and returns to the block that contains it (or returns to the code that called the block), with the exception still raised. This process repeats with increasingly broad scope until either execution reaches the outermost scope of the application or a block at some level handles the exception.

Thus, you can nest your exception handling code. That is, you can use nested blocks to define local handling for specific exceptions that overrides the handling in the surrounding block. For example:

```
try
```

```
{ statements }
  try
{ special statements }
  except
    on ESomething do
    begin
{ handling for only the special statements }
    end;
  end;
  { more statements }
except
  on ESomething do
  begin
    {handling for statements and more statements, but not special statements}
  end;
end;
try
{
  // statements
  try
  {
    // special statements
  }
  catch (const ESomething &E)
  {
    // handling for only the special statements;
  }
  // more statements
}
catch (const ESomething &E)
{
  // handling for statements and more statements, but not special statements
}
```

Note: This type of nesting is not limited to exception-handling blocks. You can also use it with finally blocks (described in Writing Finally Blocks (see page 2025)) or a mix of exception-handling and finally blocks.

See Also

Exception-handling Statements (see page 2018)

Handling Classes of Exceptions (see page 2027)

Reraising Exceptions (see page 2020)

3.2.4.6.12 Silent Exceptions

VCL applications handle most exceptions that your code doesn't specifically handle by displaying a message box that shows the message string from the exception object. You can also define "silent" exceptions that do not, by default, cause the application to show the error message.

Silent exceptions are useful when you don't intend to report an exception to the user, but you want to abort an operation. Aborting an operation is similar to using the *Break* or *Exit* procedures to break out of a block, but can break out of several nested levels of blocks.

Silent exceptions all descend from the standard exception type *EAbort*. The default exception handler for VCL applications displays the error-message dialog box for all exceptions that reach it except those descended from *EAbort*.

Note: For console applications, an error-message dialog is displayed on any unhandled *EAbort* exceptions.

There is a shortcut for raising silent exceptions. Instead of manually constructing the object, you can call the *Abort* procedure. *Abort* automatically raises an *EAbort* exception, which breaks out of the current operation without displaying an error message.

Note: There is a distinction between *Abort* and *abort*. *abort* kills the application.

The following code shows a simple example of aborting an operation. On a form containing an empty list box and a button, attach the following code to the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
var
I, J: Integer;
begin
  for I := 1 to 10 do{ loop ten times }
    for J := 1 to 10 do {loop ten times }
    begin
      ListBox1.Items.Add(IntToStr(I) + IntToStr(J));
      if I = 7 then Abort;{ abort after the 7th iteration of outer loop}
    end;
end;
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  for (int i = 1; i <= 10; i++) // loop ten times
    for (int j = 1; j <= 10; j++) // loop ten times
    {
      ListBox1->Items->Add(IntToStr(i) + IntToStr(j));
      if (i == 7)
        Abort(); // abort after 7th iteration of outer loop
    }
}
```

Note that in this example, *Abort* causes the flow of execution to break out of both the inner and outer loops, not just the inner loop.

See Also

VCL Exception Classes (see page 2028)

Default Exception Handling in VCL (see page 2029)

Defining Your Own VCL Exceptions (see page 2029)

3.2.4.6.13 Writing a finally Block (C++)

bcc32 includes extensions to the C++ language that let it use finally blocks as well. Like exception handlers, a finally block must appear directly after the **try** block, but it is introduced by the__**finally** keyword instead of **catch**.

```
try
{
  // statements that may raise an exception
}
__finally
{
    // statements that are called even if there is an exception in the try block
}
```

The application always executes any statements in the finally part, even if an exception occurs in the try block. When any code in the **try** block (or any routine called by code in the **try** block) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the finally part. After the **finally** part executes, the exception handler is called. If no exception occurs, the code in the **finally** block executes in the normal order, after all the statements in the **try** block.

The following code illustrates an event handler that uses a finally block so that when it allocates memory and generates an error, it still frees the allocated memory:

```
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  int ADividend = 0;
  void *ptr = malloc(1024); // allocate 1K of memory;
```

```
  try
    int AnInteger = 10/ADividend; // this generates an exception
  __finally
    free(ptr); // this gets called anyway, despite the exception
}
```

The statements in the finally block do not depend on an exception occurring. If no statement in the **try** part raises an exception, execution continues through the finally block.

Note: Traditional C++ code does not include support for finally

block. Instead, it tends to use destructors to handle the freeing of resources. However, when working with VCL, which is written in Delphi, **finally** blocks are an important tool because of the way VCL objects must be allocated on the heap.

Tip: You can also use `std::auto_ptr` to ensure that allocated objects are deleted.

3.2.4.6.14 Writing a Finally Block (Delphi)

Finally blocks are introduced by the keyword **finally**. They are part of a **try..finally** statement, which has the following form:

```
try
{ statements that may raise an exception}
finally
{ statements that are called even if there is an exception in the try block}
end;
```

In a **try..finally** statement, the application always executes any statements in the finally part, even if an exception occurs in the try block. When any code in the try block (or any routine called by code in the try block) raises an exception, execution halts at that point. Once an exception handler is found, execution jumps to the finally part, which is called the cleanup code. After the finally part executes, the exception handler is called. If no exception occurs, the cleanup code is executed in the normal order, after all the statements in the try block.

The following code illustrates an event handler that uses a finally block so that when it allocates memory and generates an error, it still frees the allocated memory:

```
procedure TForm1.Button1Click(Sender: TObject);
var
APointer: Pointer;
AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  try
    AnInteger := 10 div ADividend;{ this generates an exception }
  finally
    FreeMem(APointer, 1024);{ execution resumes here, despite the exception }
  end;
end;
```

The statements in the finally block do not depend on an exception occurring. If no statement in the try part raises an exception, execution continues through the finally block.

See Also

Raising an Exception (see page 2020)

Exception-handling Statements (see page 2018)

3.2.4.6.15 Writing Exception Handlers

The exception handling block appears immediately after the try block. This block incudes one or more exception handlers. An exception handler provides a specific response to a specific kind of exception. Handling an exception clears the error condition

and destroys the exception object, which allows the application to continue execution. You typically define exception handlers to allow your applications to recover from errors and continue running. Types of exceptions you might handle include attempts to open files that don't exist, writing to full disks, or calculations that exceed legal bounds. Some of these, such as "File not found," are easy to correct and retry, while others, such as running out of memory, can be more difficult for the application or the user to correct.

The application executes the statements in and exception handler only if an exception occurs during execution of the statements in the preceding try block. When a statement in the try block raises an exception, execution immediately jumps to the exception handler, where it steps through the specified exception-handling statements, until it finds a handler that applies to the current exception.

Once the application locates an exception handler that handles the exception, it executes the statement, then automatically destroys the exception object. Execution continues at the end of the current block.

The following topics provide details on writing exception handlers:

- Exception-handling Statements (see page 2018)
- Handling Classes of Exceptions (see page 2027)
- Scope of Exception Handlers (see page 2021)
- Reraising Exceptions (see page 2020)

See Also

Writing the Try Block (see page 2026)

Writing Finally Blocks (see page 2025)

3.2.4.6.16 Writing finally Blocks

An exception handler is code that handles a specific exception or exceptions that occur within a protected block of code. However, there are times when you do not need to handle the exception, but you do have code that you want to execute after the protected block, even if an exception occurs. Typically, such code handles cleanup issues, such as freeing resources that were allocated before the protected block.

By using finally blocks, you can ensure that if your application allocates resources, it also releases them, even if an exception occurs. Thus, if your application allocates memory, you can make sure it eventually releases the memory, too. If it opens a file, you can make sure it closes the file later. Under normal circumstances, you can ensure that an application frees allocated resources by including code for both allocating and freeing. When exceptions occur, however, you need to ensure that the application still executes the resource-freeing code.

Some common resources that you should always be sure to release are:

- Files
- Memory
- Windows resources or widget library resources (Qt objects)
- Objects (instances of classes in your application)

The following event handler illustrates how an exception can prevent an application from freeing memory that it allocates:

```
procedure TForm1.Button1Click(Sender: TObject);
var
APointer: Pointer;
AnInteger, ADividend: Integer;
begin
  ADividend := 0;
  GetMem(APointer, 1024);{ allocate 1K of memory }
  AnInteger := 10 div ADividend;{ this generates an exception }
  FreeMem(APointer, 1024);{ this never gets called because of the exception}
```

```
end;
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  int ADividend = 0;
  void *ptr = malloc(1024); // allocate 1K of memory;
  int AnInteger = 10/ADividend; // this generates an exception
  free(ptr); // this never gets called because of the exception
}
```

Although most errors are not that obvious, the example illustrates an important point: When an exception occurs, execution jumps out of the block, so the statement that frees the memory never gets called.

To ensure that the memory is freed, you can use a try block with a finally block.

For details on writing finally blocks, see Writing a Finally Block (see page 2024).

See Also

Writing Exception Handlers (see page 2024)

Writing the Try Block (see page 2026)

3.2.4.6.17 Writing the Try Block

The first part of a protected block is the try block. The try block contains code that can potentially raise an exception. The exception can be raised either directly in the try block, or by code that is called by statements in the try block. That is, if code in a try block calls a routine that doesn't define its own exception handler, then any exceptions raised inside that routine cause execution to pass to the exception-handler associated with the try block. Keep in mind that exceptions don't come just from your code. A call to an RTL routine or another component in your application can also raise an exception.

The following example demonstrates catching an exception thrown from a *TFileStream* object.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  fileStream: TFileStream;
begin
     try
        (* Attempt to open a non-existant file *)
        fileStream := TFileStream.Create('NOT_THERE.FILE', fmOpenRead);
                                (* Process the file contents... *)
        fileStream.Free;
     except
        on EFOpenError do ShowMessage('EFOpenError Raised');
     else
        ShowMessage('Exception Raised');
     end;
end;
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   TFileStream *fileStream;
      try {
// Attempt to open a non-existent file
         fileStream = new TFileStream("NOT_THERE.FILE", fmOpenRead);
// Process the file contents...
         delete fileStream;
      }
      catch(EFOpenError &e) {
         ShowMessage("EFOpenError Raised");
      }
      catch(...) {
         ShowMessage("Exception Raised");
      }
}
```

Using a try block makes your code easier to read. Instead of sprinkling error-handling code throughout your program, you isolate it in exception handlers so that the flow of your algorithms is more obvious.

This is especially true when performing complex calculations involving hundreds of steps, any one of which could fail if one of dozens of inputs were invalid. By using exceptions, you can spell out the normal expression of your algorithm, then provide for those exceptional cases when it doesn't apply. Without exceptions, you have to test every time to make sure you can proceed with each step in the calculation.

For details on raising exceptions from the code in your try block, see Raising an Exception (see page 2020).

See Also

Writing Exception Handlers (see page 2024)

Writing Finally Blocks (see page 2025)

3.2.4.6.18 Handling Classes of Exceptions

Exceptions are always represented by classes. As such, you usually work with a hierarchy of exception classes. For example, VCL defines the *ERangeError* exception as a descendant of *EIntError*.

When you provide an exception handler for a base exception class, it catches not only direct instances of that class, but instances of any of its descendants as well. For example, the following exception handler handles all integer math exceptions, including *ERangeError*, *EDivByZero*, and *EIntOverflow*:

```
try
{ statements that perform integer math operations }
except
  on EIntError do { special handling for integer math errors };
end;
```

```
try
{
  // statements that perform integer math operations
}
catch (EIntError &E)
{
  // special handling for integer math errors
}
```

You can combine error handlers for the base class with specific handlers for more specific (derived) exceptions. You do this by placing the **catch** statements in the order that you want them to be searched when an exception is thrown. For example, this block provides special handling for range errors, and other handling for all other integer math errors:

```
try
{ statements performing integer math }
except
  on ERangeError do { out-of-range handling };
  on EIntError do { handling for other integer math errors };
end;
```

```
try
{
  // statements performing integer math
}
catch (const ERangeError &rangeErr)
{
  // out-of-range handling
}
catch (const EIntError &intErr)
{
  // handling for other integer math errors
}
```

Note that if the handler for *EIntError* came before the handler for *ERangeError*, execution would never reach the specific handler

for *ERangeError*.

See Also

Exception-handling Statements (see page 2018)

Scope of Exception Handlers (see page 2021)

Reraising Exceptions (see page 2020)

3.2.4.6.19 VCL Exception Classes

VCL includes a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions. All VCL exception classes descend from one root object called Exception. *Exception* provides a consistent interface for applications to handle exceptions. It provides the string for the message that VCL exceptions display by default.

The following table lists a selection of the exception classes defined in VCL:

Selected exception classes

Exception class	Description
EAbort	Stops a sequence of events without displaying an error message dialog box.
EAccessViolation	Checks for invalid memory access errors.
EBitsError	Prevents invalid attempts to access a Boolean array.
EComponentError	Signals an invalid attempt to register or rename a component.
EConvertError	Indicates string or object conversion errors.
EDatabaseError	Specifies a database access error.
EDBEditError	Catches data incompatible with a specified mask.
EDivByZero	Catches integer divide-by-zero errors.
EExternalException	Signifies an unrecognized exception code.
EInOutError	Represents a file I/O error.
EIntOverflow	Specifies integer calculations whose results are too large for the allocated register.
EInvalidCast	Checks for illegal typecasting.
EInvalidGraphic	Indicates an attempt to work with an unrecognized graphic file format.
EInvalidOperation	Occurs when invalid operations are attempted on a component.
EInvalidPointer	Results from invalid pointer operations.
EMenuError	Involves a problem with menu item.
EOleCtrlError	Detects problems with linking to ActiveX controls.
EOleError	Specifies OLE automation errors.
EPrinterError	Signals a printing error.
EPropertyError	Occurs on unsuccessful attempts to set the value of a property.
ERangeError	Indicates an integer value that is too large for the declared type to which it is assigned.
ERegistryException	Specifies registry errors.
EZeroDivide	Catches floating-point divide-by-zero errors.

There are other times when you will need to create your own exception classes to handle unique situations. You can declare a

new exception class by making it a descendant of type *Exception* and creating as many constructors as you need (or copy the constructors from an existing class in the SysUtils unit).

See Also

Default Exception Handling in VCL (see page 2029)

Silent Exceptions (see page 2022)

Defining Your Own VCL Exceptions (see page 2029)

3.2.4.6.20 Default Exception Handling in VCL

If your application code does not catch and handle the exceptions that are raised, the exceptions are ultimately caught and handled by the *HandleException* method of the global *Application* object. For all exceptions but *EAbort*, *HandleException* calls the *OnException* event handler, if one exists. If there is no *OnException* event handler (and the exception is not *EAbort*), *HandleException* displays a message box with the error message associated with the exception.

There are certain circumstances where *HandleException* does not get called. Exceptions that occur before or after the execution of the application's Run method are not caught and handled by *HandleException*. When you write a callback function or a library (.dll or shared object) with functions that can be called by an external application, exceptions can escape the *Application* object. To prevent exceptions from escaping in this manner, you can insert your own call to the *HandleException* method:

```
try
{ special statements }
except
  on Exception do
  begin
    Application.HandleException(Self);{ call HandleException }
  end;
end;
try
{
  // special statements
}
catch (Exception &E)
{
  Application->HandleException(this);
}
```

Warning: Do not call *HandleException* from a thread's exception handling code.

See Also

VCL Exception Classes (see page 2028)

Silent Exceptions (see page 2022)

Defining Your Own VCL Exceptions (see page 2029)

3.2.4.6.21 Defining Your Own VCL Exceptions

Because VCL exceptions are classes, defining a new kind of exception is as simple as declaring a new class type. Although you can raise any object instance as an exception, the standard VCL exception handlers handle only exceptions that descend from *Exception*.

New exception classes should be derived from *Exception* or one of the other standard exceptions. That way, if you raise your new exception in a block of code that isn't protected by an exception handler specific to that exception, one of the standard handlers will handle it instead.

For example, consider the following declaration:

```
type
  EMyException = class(Exception);
class EMyException : public Exception
{
};
```

If you raise *EMyException* but don't provide a specific handler for it, a handler for *Exception* (or a default exception handler) will still handle it. Because the standard handling for *Exception* displays the name of the exception raised, you can see that it is your new exception that is raised.

See Also

VCL Exception Classes (see page 2028)

Default Exception Handling in VCL (see page 2029)

Silent Exceptions (see page 2022)

3.2.4.6.22 Handling Exceptions in VCL Applications

If you use VCL components or the VCL runtime library in your applications, you need to understand the VCL exception handling mechanism. Exceptions are built into many VCL classes and routines and they are thrown automatically when something unexpected occurs. Typically, these exceptions indicate programming errors that would otherwise generate a runtime error. A limited number of these classes is described in VCL Exception Classes (see page 2028).

The mechanics of handling component exceptions are no different than handling any other type of exception.

If you do not handle the exception, VCL handles it in a default manner (see page 2029). Typically, a message displays describing the type of error that occurred. While debugging your application, you can look up the exception class in online Help. The information provided will often help you to determine where the error occurred and its cause.

Certain classes of exceptions do not display an error message when caught by the default handlers. These are described in Silent Exceptions (see page 2022).

A common source of errors in components is range errors in indexed properties. For example, if a list box has three items in its list (0..2) and your application attempts to access item number 3, the list box raises a "List index out of bounds" exception.

The following event handler contains an exception handler to notify the user of invalid index access in a list box:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ListBox1.Items.Add('a string');{ add a string to list box }
  ListBox1.Items.Add('another string');{ add another string... }
  ListBox1.Items.Add('still another string');{ ...and a third string }
  try
    Caption :=

ListBox1.Items[3]

;{ set form caption to fourth string }
  except
    on EStringListError do
      ShowMessage('List box contains fewer than four strings');
  end;
end;
void __fastcall TForm1::Button1Click(TObject* Sender)
{
  ListBox1->Items->Add("a string"); // add a string to list box
  ListBox1->Items->Add("another string"); // add another string ...
  ListBox1->Items->Add("still another string"); // ... and a third string
  try
    Caption =
ListBox1->Items->Strings[3]
```

```
; // set form caption to 4th string
  catch (EStringListError &E)
    ShowMessage("List box contains fewer than four strings");
}
```

If you click the button once, the list box has only three strings, so accessing the fourth string raises an exception. Clicking a second time adds more strings to the list, so it no longer causes the exception.

In addition to handling the exceptions that VCL raises, you can define and raise your own VCL-based exception classes. This is discussed in Defining Your Own VCL Exceptions (see page 2029).

See Also

Defining Protected Blocks (see page 2017)

3.2.4.7 Types of controls

Topics

Name	Description
Value List Editors (VCL Only) (see page 2036)	*TValueListEditor* is a specialized grid for editing string lists that contain name/value pairs in the form Name=Value. The names and values are stored as a *TStrings* descendant that is the value of the *Strings* property. You can look up the value for any name using the *Values* property. *TValueListEditor* is not available for cross-platform programming. The grid contains two columns, one for the names and one for the values. By default, the Name column is named "Key" and the Value column is named "Value". You can change these defaults by setting the *TitleCaptions* property. You can omit these titles using... more (see page 2036)
Animation Control (see page 2037)	The animation component is a window that silently displays an Audio Video Interleaved (AVI) clip (VCL applications). An AVI clip is a series of bitmap frames, like a movie. Although AVI clips can have sound, animation controls work only with silent AVI clips. The files you use must be either uncompressed AVI files or AVI clips compressed using run-length encoding (RLE). Following are some of the properties of an animation component: • *ResHandle* is the Windows handle for the module that contains the AVI clip as a resource. Set *ResHandle* at runtime to the instance handle or module handle of the... more (see page 2037)
Bevels (see page 2037)	The bevel component (*TBevel*) is a line that can appear raised or lowered. Some components, such as *TPanel*, have built-in properties to create beveled borders. When such properties are unavailable, use *TBevel* to create beveled outlines, boxes, or frames.
Bitmap Buttons (see page 2038)	A bitmap button (*TBitBtn*) is a button control that presents a bitmap image on its face. • To choose a bitmap for your button, set the *Glyph* property. • Use *Kind* to automatically configure a button with a glyph and default behavior. • By default, the glyph appears to the left of any text. To move it, use the *Layout* property. • The glyph and text are automatically centered on the button. To move their position, use the *Margin* property. *Margin* determines the number of pixels between the edge of the image and the edge of the button. • By default, the image... more (see page 2038)

Button Controls (see page 2038)	Users click button controls to initiate actions. You can assign an action to a *TButto*n component by creating an *OnClick* event handler for it. Double-clicking a button at design time takes you to the button's *OnClick* event handler in the Code editor. • Set Cancel to *True* if you want the button to trigger its OnClick event when the user presses Esc. • Set Default to *True* if you want the Enter key to trigger the button's OnClick event.
Buttons and Similar Controls (see page 2038)	Aside from menus, buttons provide the most common way to initiate an action or command in an application. Button-like controls include:
Check Boxes (see page 2039)	A check box is a toggle that lets the user select an on or off state. When the choice is turned on, the check box is checked. Otherwise, the check box is blank. You create check boxes using *TCheckBox*. • Set *Checked* to *True* to make the box appear checked by default. • Set *AllowGrayed* to *True* to give the check box three possible states: checked, unchecked, and grayed. • The *State* property indicates whether the check box is checked (*cbChecked*), unchecked (*cbUnchecked*), or grayed (*cbGrayed*). **Note:** Check box controls display one of two binary states.... more (see page 2039)
Combo Boxes (see page 2039)	A combo box (*TComboBox*) combines an edit box with a scrollable list. When users enter data into the control—by typing or selecting from the list—the value of the *Text* property changes. If *AutoComplete* is enabled, the application looks for and displays the closest match in the list as the user types the data. Three types of combo boxes are: standard, drop-down (the default), and drop-down list.
Cool Bars (VCL Only) (see page 2040)	A cool bar contains child controls that can be moved and resized independently. Each control resides on an individual band. The user positions the controls by dragging the sizing grip to the left of each band. The cool bar requires version 4.70 or later of COMCTL32.DLL (usually located in the Windows\System or Windows\System32 directory) at both design time and runtime. Cool bars cannot be used in cross-platform applications. • The *Bands* property holds a collection of *TCoolBand* objects. At design time, you can add, remove, or modify bands with the Bands editor. To open the Bands editor, select the... more (see page 2040)
Draw Grids (see page 2040)	A draw grid (*TDrawGrid*) displays arbitrary data in tabular format. Write an *OnDrawCell* event handler to fill in the cells of the grid. • The *CellRect* method returns the screen coordinates of a specified cell, while the *MouseToCell* method returns the column and row of the cell at specified screen coordinates. The *Selection* property indicates the boundaries of the currently selected cells. • The *TopRow* property determines which row is currently at the top of the grid. The *LeftCol* property determines the first visible column on the left. *VisibleColCount* and *VisibleRowCount* are the number of columns and rows visible in... more (see page 2040)
Graphic Controls (see page 2040)	The following components make it easy to incorporate graphics into an application.

Group Boxes and Radio Groups (see page 2041)	A group box (*TGroupBox*) arranges related controls on a form. The most commonly grouped controls are radio buttons. After placing a group box on a form, select components from the **Tool palette** and place them in the group box. The *Caption* property contains text that labels the group box at runtime. The radio group component (*TRadioGroup*) simplifies the task of assembling radio buttons and making them work together. To add radio buttons to a radio group, edit the *Items* property in the **Object Inspector**; each string in *Items* makes a radio button appear in the... more (see page 2041)
Grouping Controls (see page 2041)	A graphical interface is easier to use when related controls and information are presented in groups. Components for grouping components include:
List Controls (see page 2042)	Lists present the user with a collection of items to select from. Several components display lists:
Header Controls (see page 2042)	A header control (*THeaderControl*) is a is a set of column headers that the user can select or resize at runtime. Edit the control's *Sections* property to add or modify headers. You can place the header sections above columns or fields. For example, header sections might be placed over a list box (*TListBox*).
Help and Hint Properties (see page 2042)	Most visual controls can display context-sensitive Help as well as fly-by hints at runtime. The *HelpContext* and *HelpFile* properties establish a Help context number and Help file for the control. The *Hint* property contains the text string that appears when the user moves the mouse pointer over a control or menu item. To enable hints, set *ShowHint* to *True*; setting *ParentShowHint* to *True* causes the control's *ShowHint* property to have the same value as its parent's.
Hot Key Controls (VCL Only) (see page 2043)	Use the hot key component (*THotKey*) to assign a keyboard shortcut that transfers focus to any control. The HotKey property contains the current key combination and the *Modifiers* property determines which keys are available for *HotKey*. The hot key component can be assigned as the *ShortCut* property of a menu item. Then, when a user enters the key combination specified by the *HotKey* and *Modifiers* properties, Windows activates the menu item.
Images (see page 2043)	The image component (*TImage*) displays a graphical image, like a bitmap, icon, or metafile. The *Picture* property determines the graphic to be displayed. Use *Center, AutoSize, Stretch*, and *Transparent* to set display options. For more information, see Overview of Graphics Programming (see page 2176).
Labels (see page 2043)	Labels display text and are usually placed next to other controls.
List Boxes and Check-list Boxes (see page 2044)	List boxes (*TListBox*) and check-list boxes display lists from which users can select one or more choices from a list of possible options. The choices are represented using text, graphics, or both. • *Items* uses a *TStrings* object to fill the control with values. • *ItemIndex* indicates which item in the list is selected. • *MultiSelect* specifies whether a user can select more than one item at a time. • *Sorted* determines whether the list is arranged alphabetically. • *Columns* specifies the number of columns in the list control. • *IntegralHeight* specifies whether the list box shows only entries that fit completely in the... more (see page 2044)

List Views (see page 2044)	List views, created using *TListView*, display lists in various formats. Use the *ViewStyle* property to choose the kind of list you want: • *vsIcon* and *vsSmallIcon* display each item as an icon with a label. Users can drag items within the list view window (VCL only). • *vsList* displays items as labeled icons that cannot be dragged. • *vsReport* displays items on separate lines with information arranged in columns. The leftmost column contains a small icon and label, and subsequent columns contain subitems specified by the application. Use the *ShowColumnHeaders* property to display headers for the columns.
Memo and Rich Edit Controls (see page 2044)	Both the *TMemo* and *TRichEdit* controls handle multiple lines of text. *TMemo* is another type of edit box that handles multiple lines of text. The lines in a memo control can extend beyond the right boundary of the edit box, or they can wrap onto the next line. You control whether the lines wrap using the *WordWrap* property. *TRichEdit* is a memo control that supports rich text formatting, printing, searching, and drag-and-drop of text. It allows you to specify font properties, alignment, tabs, indentation, and numbering. **Note:** The rich edit control is available for VCL applications only. In addition to... more (see page 2044)
Page Controls (see page 2045)	The page control component (*TPageControl*) is a page set suitable for multipage dialog boxes. A page control displays multiple overlapping pages that are *TTabSheet* objects. A page is selected in the user interface by clicking a tab on top of the control. To create a new page in a page control at design time, right-click the control and choose New Page. At runtime, you add new pages by creating the object for the page and setting its *PageControl* property:
Paint Boxes (see page 2045)	The paint box (*TPaintBox*) allows your application to draw on a form. Write an *OnPaint* event handler to render an image directly on the paint box's *Canvas*. Drawing outside the boundaries of the paint box is prevented. For more information, see Overview of Graphics Programming (see page 2176).
Panels (see page 2046)	The *TPanel* component provides a generic container for other controls. Panels are typically used to visually group components together on a form. Panels can be aligned with the form to maintain the same relative position when the form is resized. The *BorderWidth* property determines the width, in pixels, of the border around a panel. You can also place other controls onto a panel and use the *Align* property to ensure proper positioning of all the controls in the group on the form. You can make a panel alTop aligned so that its position will remain in place even if the... more (see page 2046)
Progress Bars (see page 2046)	When your application performs a time-consuming operation, you can use a progress bar (*TProgressBar*) to show how much of the task is completed. A progress bar displays a dotted line that grows from left to right. The *Position* property tracks the length of the dotted line. *Max* and *Min* determine the range of *Position*. To make the line grow, increment *Position* by calling the *StepBy* or *StepIt* method. The *Step* property determines the increment used by *StepIt*.
Radio Buttons (see page 2046)	Radio buttons, also called option buttons, present a set of mutually exclusive choices. You can create individual radio buttons using *TRadioButton* or use the *radio group* component (*TRadioGroup*) to arrange radio buttons into groups automatically. You can group radio buttons to let the user select one from a limited set of choices. See Grouping Controls (see page 2041) for more information. A selected radio button is displayed as a circle filled in the middle. When not selected, the radio button shows an empty circle. Assign the value *True* or *False* to the Checked property to change the radio button's visual state.... more (see page 2046)

Scroll Bars (see page 2046)	The scroll bar component creates a scroll bar that you can use to scroll the contents of a window, form, or other control. In the *OnScroll* event handler, you write code that determines how the control behaves when the user moves the scroll bar. The scroll bar component is not used very often, because many visual components include scroll bars of their own and thus don't require additional coding. For example, *TForm* has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable region within a form, use *TScrollBox*.
Scroll Boxes (see page 2047)	Scroll boxes (*TScrollBox*) create scrolling areas within a form. Applications often need to display more information than will fit in a particular area. Some controls—such as list boxes, memos, and forms themselves—can automatically scroll their contents. Another use of scroll boxes is to create multiple scrolling areas (views) in a window. Views are common in commercial word-processor, spreadsheet, and project management applications. Scroll boxes give you the additional flexibility to define arbitrary scrolling subregions of a form. Like panels and group boxes, scroll boxes contain other controls, such as *TButton* and *TCheckBox* objects. But a scroll box is... more (see page 2047)
Shapes (see page 2047)	The shape component displays a geometric shape. It is a nonwindowed control and therefore, cannot receive user input. The *Shape* property determines which shape the control assumes. To change the shape's color or add a pattern, use the *Brush* property, which holds a *TBrush* object. How the shape is painted depends on the *Color* and *Style* properties of *TBrush*.
Specialized Input Controls (see page 2047)	The following components provide additional ways of capturing input.
Speed Buttons (see page 2048)	Speed buttons (TSpeedButton), which usually have images on their faces, can function in groups. They are commonly used with panels to create toolbars. • To make speed buttons act as a group, give the *GroupIndex* property of all the buttons the same nonzero value. • By default, speed buttons appear in an up (unselected) state. To initially display a speed button as selected, set the *Down* property to *True*. • If *AllowAllUp* is *True*, all of the speed buttons in a group can be unselected. Set *AllowAllUp* to *False* if you want a group of buttons to act like a radio... more (see page 2048)
Splitter Controls (see page 2048)	A splitter (*TSplitter*) placed between aligned controls allows users to resize the controls. Used with components like panels and group boxes, splitters let you divide a form into several panes with multiple controls on each pane. After placing a panel or other control on a form, add a splitter with the same alignment as the control. The last control should be client-aligned, so that it fills up the remaining space when the others are resized. For example, you can place a panel at the left edge of a form, set its *Alignment to alLeft*, then place a... more (see page 2048)
Status Bars (see page 2048)	Although you can use a panel to make a status bar, it is simpler to use the *TStatusBar* component. By default, the status bar's *Align* property is set to *alBottom*, which takes care of both position and size. If you only want to display one text string at a time in the status bar, set its *SimplePanel* property to *True* and use the *SimpleText* property to control the text displayed in the status bar. You can also divide a status bar into several text areas, called panels. To create panels, edit the *Panels* property in the **Object Inspector**,... more (see page 2048)
String Grids (see page 2049)	The string grid component is a descendant of *TDrawGrid* that adds specialized functionality to simplify the display of strings. The *Cells* property lists the strings for each cell in the grid; the *Objects* property lists objects associated with each string. All the strings and associated objects for a particular column or row can be accessed through the *Cols* or *Rows* property.
Tab Controls (see page 2049)	The tab control component (*TTabControl*) creates a set of tabs that look like notebook dividers. You can create tabs by editing the *Tabs* property in the **Object Inspector**; each string in *Tabs* represents a tab. The tab control is a single panel with one set of components on it. To change the appearance of the control when the tabs are clicked, you need to write an *OnChange* event handler. To create a multipage dialog box, use a page control instead.

Grids (see page 2049)	Grids display information in rows and columns. If you're writing a database application, use the TDBCtrlGrid or TDBCtrlGrid component. Otherwise, use a standard draw grid or string grid.
Edit Controls (see page 2049)	Edit controls display text to the user and allow the user to enter text. The type of control used for this purpose depends on the size and format of the information.
Toolbar Controls (see page 2050)	Toolbars provide an easy way to arrange and manage visual controls. You can create a toolbar out of a panel component and speed buttons, or you can use the *TToolBar* component, then right-click and choose New Button to add buttons to the toolbar. The *TToolBar* component has several advantages: buttons on a toolbar automatically maintain uniform dimensions and spacing; other controls maintain their relative position and height; controls can automatically wrap around to start a new row when they do not fit horizontally; and *TToolBar* offers display options like transparency, pop-up borders, and spaces and dividers to group controls. You... more (see page 2050)
Track Bars (see page 2051)	A track bar can set integer values on a continuous range. It is useful for adjusting properties like color, volume and brightness. The user moves the slide indicator by dragging it to a particular location or clicking within the bar. • Use the *Max* and *Min* properties to set the upper and lower range of the track bar. • Use *SelEnd* and *SelStart* to highlight a selection range. • The *Orientation* property determines whether the track bar is vertical or horizontal. • By default, a track bar has one row of ticks along the bottom. Use the *TickMarks* property to change their location. To... more (see page 2051)
Tree Views (see page 2051)	A tree view *(TTreeView)* displays items in an indented outline. The control provides buttons that allow nodes to be expanded and collapsed. You can include icons with items' text labels and display different icons to indicate whether a node is expanded or collapsed. You can also include graphics, such as check boxes, that reflect state information about the items. • *Indent* sets the number of pixels horizontally separating items from their parents. • *ShowButtons* enables the display of "+" and "–" buttons to indicate whether an item can be expanded. • *ShowLines* enables display of connecting lines to show hierarchical relationships... more (see page 2051)
Up-down Controls (VCL Only) (see page 2052)	In VCL applications only, an up-down control (*TUpDown*) consists of a pair of arrow buttons that allow users to change an integer value in fixed increments. The current value is given by the *Position* property; the increment, which defaults to 1, is specified by the *Increment* property. Use the *Associate* property to attach another component (such as an edit control) to the up-down control.
Display Controls (see page 2052)	There are many ways to provide users with information about the state of an application. For example, some components—including *TForm*—have a *Caption* property that can be set at runtime. You can also create dialog boxes to display messages. In addition, the following components are especially useful for providing visual feedback at runtime to identify the object.
Text Controls (see page 2052)	Many applications use text controls to display text to the user. You can use: • Edit controls (see page 2049), which allow the user to add text. • Text viewing controls (see page 2052) and labels (see page 2043), which do not allow user to add text.
Text Viewing Controls (see page 2052)	

3.2.4.7.1 Value List Editors (VCL Only)

TValueListEditor is a specialized grid for editing string lists that contain name/value pairs in the form Name=Value. The names and values are stored as a *TStrings* descendant that is the value of the *Strings* property. You can look up the value for any name

using the *Values* property. *TValueListEditor* is not available for cross-platform programming.

The grid contains two columns, one for the names and one for the values. By default, the Name column is named "Key" and the Value column is named "Value". You can change these defaults by setting the *TitleCaptions* property. You can omit these titles using the *DisplayOptions* property (which also controls resize when you resize the control.)

You can control whether users can edit the Name column using the *KeyOptions* property. *KeyOptions* contains separate options to allow editing, adding new names, deleting names, and controlling whether new names must be unique.

You can control how users edit the entries in the Value column using the *ItemProps* property. Each item has a separate *TItemProp* object that lets you

- Supply an edit mask to limit the valid input.
- Specify a maximum length for values.
- Mark the value as read-only.
- Specify that the value list editor displays a drop-down arrow that opens a pick list of values from which the user can choose or an ellipsis button that triggers an event you can use for displaying a dialog in which users enter values.

If you specify that there is a drop-down arrow, you must supply the list of values from which the user chooses. These can be a static list (the *PickList* property of the *TItemProp* object) or they can be dynamically added at runtime using the value list editor's *OnGetPickList* event. You can also combine these approaches and have a static list that the *OnGetPickList* event handler modifies.

If you specify that there is an ellipsis button, you must supply the response that occurs when the user clicks that button (including the setting of a value, if appropriate). You provide this response by writing an *OnEditButtonClick* event handler.

See Also

TValueListEditor

3.2.4.7.2 Animation Control

The animation component is a window that silently displays an Audio Video Interleaved (AVI) clip (VCL applications). An AVI clip is a series of bitmap frames, like a movie. Although AVI clips can have sound, animation controls work only with silent AVI clips. The files you use must be either uncompressed AVI files or AVI clips compressed using run-length encoding (RLE).

Following are some of the properties of an animation component:

- *ResHandle* is the Windows handle for the module that contains the AVI clip as a resource. Set *ResHandle* at runtime to the instance handle or module handle of the module that includes the animation resource. After setting *ResHandle*, set the *ResID* or *ResName* property to specify which resource in the indicated module is the AVI clip that should be displayed by the animation control.
- Set *AutoSize* to *True* to have the animation control adjust its size to the size of the frames in the AVI clip.
- *StartFrame* and *StopFrame* specify in which frames to start and stop the clip.
- Set *CommonAVI* to display one of the common Windows AVI clips provided in Shell32.DLL.
- Specify when to start and interrupt the animation by setting the *Active* property to *True* and *False*, respectively, and how many repetitions to play by setting the *Repetitions* property.
- The *Timers* property lets you display the frames using a timer. This is useful for synchronizing the animation sequence with other actions, such as playing a sound track.

See Also

TAnimate

3.2.4.7.3 Bevels

The bevel component (*TBevel*) is a line that can appear raised or lowered. Some components, such as *TPanel*, have built-in

properties to create beveled borders. When such properties are unavailable, use *TBevel* to create beveled outlines, boxes, or frames.

See Also

TBevel

3.2.4.7.4 Bitmap Buttons

A bitmap button (*TBitBtn*) is a button control that presents a bitmap image on its face.

- To choose a bitmap for your button, set the *Glyph* property.
- Use *Kind* to automatically configure a button with a glyph and default behavior.
- By default, the glyph appears to the left of any text. To move it, use the *Layout* property.
- The glyph and text are automatically centered on the button. To move their position, use the *Margin* property. *Margin* determines the number of pixels between the edge of the image and the edge of the button.
- By default, the image and the text are separated by 4 pixels. Use *Spacing* to increase or decrease the distance.
- Bitmap buttons can have 3 states: up, down, and held down. Set the *NumGlyphs* property to 3 to show a different bitmap for each state.

See Also

TBitBtn

3.2.4.7.5 Button Controls

Users click button controls to initiate actions. You can assign an action to a *TButto*n component by creating an *OnClick* event handler for it. Double-clicking a button at design time takes you to the button's *OnClick* event handler in the Code editor.

- Set Cancel to *True* if you want the button to trigger its OnClick event when the user presses Esc.
- Set Default to *True* if you want the Enter key to trigger the button's OnClick event.

See Also

TButton

3.2.4.7.6 Buttons and Similar Controls

Aside from menus, buttons provide the most common way to initiate an action or command in an application. Button-like controls include:

Use this component:	To do this:
TButton	Present command choices on buttons with text
TBitBtn	Present command choices on buttons with text and glyphs
TSpeedButton	Create grouped toolbar buttons
TCheckBox	Present on/off options
TRadioButton	Present a set of mutually exclusive choices
TToolBar	Arrange tool buttons and other controls in rows and automatically adjust their sizes and positions
TCoolBar	Display a collection of windowed controls within movable, resizable bands (VCL only)

Action lists let you centralize responses to user commands (actions) for objects such as menus and buttons that respond to

those commands. See Using action lists (see page 2007) for details on how to use action lists with buttons, toolbars, and menus.

You can custom draw buttons individually or application wide. See Developing the user interface. (see page 1983)

See Also

TButton

TBitBtn

TSpeedButton

TCheckBox

TRadioButton

TToolBar

TCoolBar

3.2.4.7.7 Check Boxes

A check box is a toggle that lets the user select an on or off state. When the choice is turned on, the check box is checked. Otherwise, the check box is blank. You create check boxes using *TCheckBox*.

- Set *Checked* to *True* to make the box appear checked by default.
- Set *AllowGrayed* to *True* to give the check box three possible states: checked, unchecked, and grayed.
- The *State* property indicates whether the check box is checked (*cbChecked*), unchecked (*cbUnchecked*), or grayed (*cbGrayed*).

  Note: Check box controls display one of two binary states. The indeterminate state is used when other settings make it impossible to determine the current value for the check box.

See Also

TCheckBox

TDBCheckBox

3.2.4.7.8 Combo Boxes

A combo box (*TComboBox*) combines an edit box with a scrollable list. When users enter data into the control—by typing or selecting from the list—the value of the *Text* property changes. If *AutoComplete* is enabled, the application looks for and displays the closest match in the list as the user types the data.

Three types of combo boxes are: standard, drop-down (the default), and drop-down list.

To create a combo box

1. Set the *Style* property to select the type of combo box you need:

- Use *csDropDown* to create an edit box with a drop-down list. Use *csDropDownList* to make the edit box read-only (forcing users to choose from the list).
- Use *csOwnerDrawFixed* or *csOwnerDrawVariable* to create *owner-draw* combo boxes that display items graphically or in varying heights. For information on owner-draw controls, see Adding Graphics to Controls (see page 2154).
- Use *csSimple* to create a combo box with a fixed list that does not close. Be sure to resize the combo box so that the list items are displayed (VCL only).

2. Set the *DropDownCount* property to change the number of items displayed in the list.

See Also

TComboBox

3.2.4.7.9 Cool Bars (VCL Only)

A cool bar contains child controls that can be moved and resized independently. Each control resides on an individual band. The user positions the controls by dragging the sizing grip to the left of each band.

The cool bar requires version 4.70 or later of COMCTL32.DLL (usually located in the Windows\System or Windows\System32 directory) at both design time and runtime. Cool bars cannot be used in cross-platform applications.

- The *Bands* property holds a collection of *TCoolBand* objects. At design time, you can add, remove, or modify bands with the Bands editor. To open the Bands editor, select the *Bands* property in the **Object Inspector**, then double-click in the Value column to the right, or click the ellipsis (...) button. You can also create bands by adding new windowed controls from the palette.
- The *FixedOrder* property determines whether users can reorder the bands.
- The *FixedSize* property determines whether the bands maintain a uniform height.

See Also

TCoolBar

3.2.4.7.10 Draw Grids

A draw grid (*TDrawGrid*) displays arbitrary data in tabular format. Write an *OnDrawCell* event handler to fill in the cells of the grid.

- The *CellRect* method returns the screen coordinates of a specified cell, while the *MouseToCell* method returns the column and row of the cell at specified screen coordinates. The *Selection* property indicates the boundaries of the currently selected cells.
- The *TopRow* property determines which row is currently at the top of the grid. The *LeftCol* property determines the first visible column on the left. *VisibleColCount* and *VisibleRowCount* are the number of columns and rows visible in the grid.
- You can change the width or height of a column or row with the *ColWidths* and *RowHeights* properties. Set the width of the grid lines with the *GridLineWidth* property. Add scroll bars to the grid with the *ScrollBars* property.
- You can choose to have fixed or non-scrolling columns and rows with the *FixedCols* and *FixedRows* properties. Assign a color to the fixed columns and rows with the *FixedColor* property.
- The *Options*, *DefaultColWidth*, and *DefaultRowHeight* properties also affect the appearance and behavior of the grid.

See Also

TDrawGrid

3.2.4.7.11 Graphic Controls

The following components make it easy to incorporate graphics into an application.

Use this component:	To display:
TImage	Graphics files
TShape	Geometric shapes
TBevel	3-D lines and frames
TPaintBox	Graphics drawn by your program at runtime
TAnimate	AVI files (VCL applications)

Notice that these include common paint routines (*Repaint, Invalidate*, and so on) that never need to receive focus.

To create a graphic control, see Creating a graphic control (see page 1220).

See Also

TGraphic

3.2.4.7.12 Group Boxes and Radio Groups

A group box (*TGroupBox*) arranges related controls on a form. The most commonly grouped controls are radio buttons. After placing a group box on a form, select components from the **Tool palette** and place them in the group box. The *Caption* property contains text that labels the group box at runtime.

The radio group component (*TRadioGroup*) simplifies the task of assembling radio buttons and making them work together. To add radio buttons to a radio group, edit the *Items* property in the **Object Inspector**; each string in *Items* makes a radio button appear in the group box with the string as its caption. The value of the *ItemIndex* property determines which radio button is currently selected. Display the radio buttons in a single column or in multiple columns by setting the value of the *Columns* property. To respace the buttons, resize the radio group component.

See Also

TGroupBox

TRadioGroup

TRadioButton

3.2.4.7.13 Grouping Controls

A graphical interface is easier to use when related controls and information are presented in groups. Components for grouping components include:

Use this component:	When you want this:
TGroupBox	A standard group box with a title
TRadioGroup	A simple group of radio buttons
TPanel	A more visually flexible group of controls
TScrollBox	A scrollable region containing controls
TTabControl	A set of mutually exclusive notebook-style tabs
TPageControl	A set of mutually exclusive notebook-style tabs with corresponding pages, each of which may contain other controls
THeaderControl	Resizable column headers

See Also

TGroupBox

TRadioGroup

TPanel

TScrollBox

TTabControl

TPageControl

THeaderControl

Splitter Control (see page 2048)

3.2.4.7.14 List Controls

Lists present the user with a collection of items to select from. Several components display lists:

Use this component:	To display:
TListBox	A list of text strings
TCheckListBox	A list with a check box in front of each item
TComboBox	An edit box with a scrollable drop-down list
TTreeView	A hierarchical list
TListView	A list of (draggable) items with optional icons, columns, and headings
TDateTimePicker	A list box for entering dates or times (VCL applications only)
TMonthCalendar	A calendar for selecting dates (VCL applications only)

Use the nonvisual *TStringList* and *TImageList* components to manage sets of strings and images. For more information about string lists, see Working with string lists (see page 2140).

See Also

TImageList

TStringList

3.2.4.7.15 Header Controls

A header control (*THeaderControl*) is a is a set of column headers that the user can select or resize at runtime. Edit the control's *Sections* property to add or modify headers. You can place the header sections above columns or fields. For example, header sections might be placed over a list box (*TListBox*).

See Also

THeaderControl

3.2.4.7.16 Help and Hint Properties

Most visual controls can display context-sensitive Help as well as fly-by hints at runtime. The *HelpContext* and *HelpFile* properties establish a Help context number and Help file for the control.

The *Hint* property contains the text string that appears when the user moves the mouse pointer over a control or menu item. To enable hints, set *ShowHint* to *True*; setting *ParentShowHint* to *True* causes the control's *ShowHint* property to have the same value as its parent's.

See Also

THintWindow

TWinControl

HelpContext

3.2.4.7.17 Hot Key Controls (VCL Only)

Use the hot key component (*THotKey*) to assign a keyboard shortcut that transfers focus to any control. The HotKey property contains the current key combination and the *Modifiers* property determines which keys are available for *HotKey*.

The hot key component can be assigned as the *ShortCut* property of a menu item. Then, when a user enters the key combination specified by the *HotKey* and *Modifiers* properties, Windows activates the menu item.

See Also

THotKey

3.2.4.7.18 Images

The image component (*TImage*) displays a graphical image, like a bitmap, icon, or metafile. The *Picture* property determines the graphic to be displayed. Use *Center, AutoSize, Stretch*, and *Transparent* to set display options. For more information, see Overview of Graphics Programming (see page 2176).

See Also

TImage

3.2.4.7.19 Labels

Labels display text and are usually placed next to other controls.

Use this component:	When you want users to do this:
TLabel	Display text on a nonwindowed control.
TStaticText	Display text on a windowed control.

You place a label on a form when you need to identify or annotate another component such as an edit box or when you want to include text on a form. The standard label component, *TLabel*, is a non-windowed control, so it cannot receive focus; when you need a label with a window handle, use *TStaticText* instead.

Label properties include the following:

- *Caption* contains the text string for the label.
- *Font*, *Color*, and other properties determine the appearance of the label. Each label can use only one typeface, size, and color.
- *FocusControl* links the label to another control on the form. If *Caption* includes an accelerator key, the control specified by *FocusControl* receives focus when the user presses the accelerator key.
- *ShowAccelChar* determines whether the label can display an underlined accelerator character. If *ShowAccelChar* is *True*, any character preceded by an ampersand (&) appears underlined and enables an accelerator key.
- *Transparent* determines whether items under the label (such as graphics) are visible.

Labels usually display read-only static text that cannot be changed by the application user. The application can change the text while it is executing by assigning a new value to the *Caption* property. To add a text object to a form that a user can scroll or edit, use *TEdit*.

See Also

TLabel

TStaticText

3.2.4.7.20 List Boxes and Check-list Boxes

List boxes (*TListBox*) and check-list boxes display lists from which users can select one or more choices from a list of possible options. The choices are represented using text, graphics, or both.

- *Items* uses a *TStrings* object to fill the control with values.
- *ItemIndex* indicates which item in the list is selected.
- *MultiSelect* specifies whether a user can select more than one item at a time.
- *Sorted* determines whether the list is arranged alphabetically.
- *Columns* specifies the number of columns in the list control.
- *IntegralHeight* specifies whether the list box shows only entries that fit completely in the vertical space (VCL only).
- *ItemHeight* specifies the height of each item in pixels. The *Style* property can cause *ItemHeight* to be ignored.
- The *Style* property determines how a list control displays its items. By default, items are displayed as strings. By changing the value of *Style*, you can create *owner-draw* list boxes that display items graphically or in varying heights. For information on owner-draw controls, see Adding Graphics to Controls (see page 2154).

To create a simple list box

1. Within your project, drop a list box component from the **Tool palette** onto a form.
2. Size the list box and set its alignment as needed.
3. Double-click the right side of the *Items* property or choose the ellipsis button to display the String List Editor.
4. Use the editor to enter free form text arranged in lines for the contents of the list box.
5. Then choose OK.

To let users select multiple items in the list box, you can use the *ExtendedSelect* and *MultiSelect* properties.

See Also

TListBox

TCheckListBox

3.2.4.7.21 List Views

List views, created using *TListView*, display lists in various formats. Use the *ViewStyle* property to choose the kind of list you want:

- *vsIcon* and *vsSmallIcon* display each item as an icon with a label. Users can drag items within the list view window (VCL only).
- *vsList* displays items as labeled icons that cannot be dragged.
- *vsReport* displays items on separate lines with information arranged in columns. The leftmost column contains a small icon and label, and subsequent columns contain subitems specified by the application. Use the *ShowColumnHeaders* property to display headers for the columns.

See Also

TListView

3.2.4.7.22 Memo and Rich Edit Controls

Both the *TMemo* and *TRichEdit* controls handle multiple lines of text.

TMemo is another type of edit box that handles multiple lines of text. The lines in a memo control can extend beyond the right boundary of the edit box, or they can wrap onto the next line. You control whether the lines wrap using the *WordWrap* property.

TRichEdit is a memo control that supports rich text formatting, printing, searching, and drag-and-drop of text. It allows you to specify font properties, alignment, tabs, indentation, and numbering.

Note: The rich edit control is available for VCL applications only.

In addition to the properties that all edit controls (see page 2049) have, memo and rich edit controls include other properties, such as the following:

- *Alignment* specifies how text is aligned (left, right, or center) in the component.
- The *Text* property contains the text in the control. Your application can tell if the text changes by checking the Modified property.
- *Lines* contains the text as a list of strings.
- *OEMConvert* determines whether the text is temporarily converted from ANSI to OEM as it is entered. This is useful for validating file names (VCL only).
- *WordWrap* determines whether the text will wrap at the right margin.
- *WantReturns* determines whether the user can insert hard returns in the text.
- *WantTabs* determines whether the user can insert tabs in the text.
- *AutoSelect* determines whether the text is automatically selected (highlighted) when the control becomes active.

At runtime, you can select all the text in the memo with the *SelectAll* method.

Note: Under Windows 9x both Memo and Rich Edit controls are limited to storing 64 kb of data.

See Also

TMemo

TRichEdit

3.2.4.7.23 Page Controls

The page control component (*TPageControl*) is a page set suitable for multipage dialog boxes. A page control displays multiple overlapping pages that are *TTabSheet* objects. A page is selected in the user interface by clicking a tab on top of the control.

To create a new page in a page control at design time, right-click the control and choose New Page. At runtime, you add new pages by creating the object for the page and setting its *PageControl* property:

```
NewTabSheet = TTabSheet.Create(PageControl1);
NewTabSheet.PageControl := PageControl1;
TTabSheet *pTabSheet = new TTabSheet(PageControl1);
pTabSheet->PageControl = PageControl1;
```

To access the active page, use the *ActivePage* property. To change the active page, you can set either the *ActivePage* or the *ActivePageIndex* property.

See Also

TPageControl

3.2.4.7.24 Paint Boxes

The paint box (*TPaintBox*) allows your application to draw on a form. Write an *OnPaint* event handler to render an image directly on the paint box's *Canvas*. Drawing outside the boundaries of the paint box is prevented. For more information, see Overview of Graphics Programming (see page 2176).

See Also

TPaintBox

3.2.4.7.25 Panels

The *TPanel* component provides a generic container for other controls. Panels are typically used to visually group components together on a form. Panels can be aligned with the form to maintain the same relative position when the form is resized. The *BorderWidth* property determines the width, in pixels, of the border around a panel.

You can also place other controls onto a panel and use the *Align* property to ensure proper positioning of all the controls in the group on the form. You can make a panel alTop aligned so that its position will remain in place even if the form is resized.

The look of the panel can be changed to a raised or lowered look by using the *BevelOuter* and *BevelInner* properties. You can vary the values of these properties to create different visual 3-D effects. Note that if you merely want a raised or lowered bevel, you can use the less resource intensive *TBevel* control instead.

You can also use one or more panels to build various status bars or information display areas.

See Also

TPanel

Splitter Control (see page 2048)

3.2.4.7.26 Progress Bars

When your application performs a time-consuming operation, you can use a progress bar (*TProgressBar*) to show how much of the task is completed. A progress bar displays a dotted line that grows from left to right.

The *Position* property tracks the length of the dotted line. *Max* and *Min* determine the range of *Position*. To make the line grow, increment *Position* by calling the *StepBy* or *StepIt* method. The *Step* property determines the increment used by *StepIt*.

See Also

TProgressBar

Track Bar Controls (see page 2051)

3.2.4.7.27 Radio Buttons

Radio buttons, also called option buttons, present a set of mutually exclusive choices. You can create individual radio buttons using *TRadioButton* or use the *radio group* component (*TRadioGroup*) to arrange radio buttons into groups automatically. You can group radio buttons to let the user select one from a limited set of choices. See Grouping Controls (see page 2041) for more information.

A selected radio button is displayed as a circle filled in the middle. When not selected, the radio button shows an empty circle. Assign the value *True* or *False* to the Checked property to change the radio button's visual state.

See Also

TRadioButton

3.2.4.7.28 Scroll Bars

The scroll bar component creates a scroll bar that you can use to scroll the contents of a window, form, or other control. In the *OnScroll* event handler, you write code that determines how the control behaves when the user moves the scroll bar.

The scroll bar component is not used very often, because many visual components include scroll bars of their own and thus don't require additional coding. For example, *TForm* has *VertScrollBar* and *HorzScrollBar* properties that automatically configure scroll bars on the form. To create a scrollable region within a form, use *TScrollBox*.

See Also

TScrollBar

Header Controls (see page 2042)

3.2.4.7.29 Scroll Boxes

Scroll boxes (*TScrollBox*) create scrolling areas within a form. Applications often need to display more information than will fit in a particular area. Some controls—such as list boxes, memos, and forms themselves—can automatically scroll their contents.

Another use of scroll boxes is to create multiple scrolling areas (views) in a window. Views are common in commercial word-processor, spreadsheet, and project management applications. Scroll boxes give you the additional flexibility to define arbitrary scrolling subregions of a form.

Like panels and group boxes, scroll boxes contain other controls, such as *TButton* and *TCheckBox* objects. But a scroll box is normally invisible. If the controls in the scroll box cannot fit in its visible area, the scroll box automatically displays scroll bars.

Another use of a scroll box is to restrict scrolling in areas of a window, such as a toolbar or status bar (*TPanel* components). To prevent a toolbar and status bar from scrolling, hide the scroll bars, and then position a scroll box in the client area of the window between the toolbar and status bar. The scroll bars associated with the scroll box will appear to belong to the window, but will scroll only the area inside the scroll box.

See Also

TScrollBox

3.2.4.7.30 Shapes

The shape component displays a geometric shape. It is a nonwindowed control and therefore, cannot receive user input. The *Shape* property determines which shape the control assumes. To change the shape's color or add a pattern, use the *Brush* property, which holds a *TBrush* object. How the shape is painted depends on the *Color* and *Style* properties of *TBrush*.

See Also

TShape

TBrush

3.2.4.7.31 Specialized Input Controls

The following components provide additional ways of capturing input.

Use this component:	When you want users to do this:
TScrollBar	Select values on a continuous range
TTrackBar	Select values on a continuous range (more visually effective than a scroll bar)
TUpDown	Select a value from a spinner attached to an edit component (VCL applications only)
THotKey	Enter `Ctrl/ Shift/ Alt` keyboard sequences (VCL applications only)

3.2.4.7.32 Speed Buttons

Speed buttons (TSpeedButton), which usually have images on their faces, can function in groups. They are commonly used with panels to create toolbars.

- To make speed buttons act as a group, give the *GroupIndex* property of all the buttons the same nonzero value.
- By default, speed buttons appear in an up (unselected) state. To initially display a speed button as selected, set the *Down* property to *True*.
- If *AllowAllUp* is *True*, all of the speed buttons in a group can be unselected. Set *AllowAllUp* to *False* if you want a group of buttons to act like a radio group.

See Also

TSpeedButton

Adding a Toolbar Using a Panel Component (see page 1969)

Adding a Speed Button to a Panel (see page 1969)

Assigning a Speed Button's Glyph (see page 1972)

Setting the Initial Condition of a Speed Button (see page 2004)

Creating a Group of Speed Buttons (see page 1979)

3.2.4.7.33 Splitter Controls

A splitter (*TSplitter*) placed between aligned controls allows users to resize the controls. Used with components like panels and group boxes, splitters let you divide a form into several panes with multiple controls on each pane.

After placing a panel or other control on a form, add a splitter with the same alignment as the control. The last control should be client-aligned, so that it fills up the remaining space when the others are resized. For example, you can place a panel at the left edge of a form, set its *Alignment to alLeft*, then place a splitter (also aligned to *alLeft*) to the right of the panel, and finally place another panel (aligned to *alLeft* or *alClient*) to the right of the splitter.

Set *MinSize* to specify a minimum size the splitter must leave when resizing its neighboring control. Set *Beveled* to *True* to give the splitter's edge a 3D look.

See Also

TSplitter

Grouping Controls (see page 2041)

3.2.4.7.34 Status Bars

Although you can use a panel to make a status bar, it is simpler to use the *TStatusBar* component. By default, the status bar's *Align* property is set to *alBottom*, which takes care of both position and size.

If you only want to display one text string at a time in the status bar, set its *SimplePanel* property to *True* and use the *SimpleText* property to control the text displayed in the status bar.

You can also divide a status bar into several text areas, called panels. To create panels, edit the *Panels* property in the **Object Inspector**, setting each panel's *Width*, *Alignment*, and *Text* properties from the Panels editor. Each panel's *Text* property contains the text displayed in the panel.

See Also

TStatusBar

Panels (see page 2046)

3.2.4.7.35 String Grids

The string grid component is a descendant of *TDrawGrid* that adds specialized functionality to simplify the display of strings. The *Cells* property lists the strings for each cell in the grid; the *Objects* property lists objects associated with each string. All the strings and associated objects for a particular column or row can be accessed through the *Cols* or *Rows* property.

See Also

TStringGrid

3.2.4.7.36 Tab Controls

The tab control component (*TTabControl*) creates a set of tabs that look like notebook dividers. You can create tabs by editing the *Tabs* property in the **Object Inspector**; each string in *Tabs* represents a tab. The tab control is a single panel with one set of components on it. To change the appearance of the control when the tabs are clicked, you need to write an *OnChange* event handler. To create a multipage dialog box, use a page control instead.

See Also

TTabControl

TPageControl

3.2.4.7.37 Grids

Grids display information in rows and columns. If you're writing a database application, use the TDBCtrlGrid or TDBCtrlGrid component. Otherwise, use a standard draw grid or string grid.

See Also

TDrawGrid

TStringGrid

TDBGrid

3.2.4.7.38 Edit Controls

Edit controls display text to the user and allow the user to enter text. The type of control used for this purpose depends on the size and format of the information.

Use this component:	When you want users to do this:
TEdit	Edit a single line of text.
TMemo	Edit multiple lines of text.
TMaskEdit	Adhere to a particular format, such as a postal code or phone number.
TRichEdit	Edit multiple lines of text using rich text format (VCL only).

TEdit and *TMaskEdit* are simple edit controls that include a single line text edit box in which you can type information. When the

edit box has focus, a blinking insertion point appears.

You can include text in the edit box by assigning a string value to its *Text* property. You control the appearance of the text in the edit box by assigning values to its *Font* property. You can specify the typeface, size, color, and attributes of the font. The attributes affect all of the text in the edit box and cannot be applied to individual characters.

An edit box can be designed to change its size depending on the size of the font it contains. You do this by setting the *AutoSize* property to *True*. You can limit the number of characters an edit box can contain by assigning a value to the *MaxLength* property.

TMaskEdit is a special edit control that validates the text entered against a mask that encodes the valid forms the text can take. The mask can also format the text that is displayed to the user.

TMemo and TRichEdit (see page 2044)*controls allow the user to add several lines of text.*

Edit controls have some of the following important properties:

Edit control properties

Property	Description
Text	Determines the text that appears in the edit box or memo control.
Font	Controls the attributes of text written in the edit box or memo control.
AutoSize	Enables the edit box to dynamically change its height depending on the currently selected font.
ReadOnly	Specifies whether the user is allowed to change the text.
MaxLength	Limits the number of characters in simple edit controls.
SelText	Contains the currently selected (highlighted) part of the text.
SelStart, SelLength	Indicate the position and length of the selected part of the text.

See Also

TEdit

TMemo

TMaskEdit

TRichEdit

Working with Text in Controls (see page 2168)

Edit Controls

3.2.4.7.39 Toolbar Controls

Toolbars provide an easy way to arrange and manage visual controls. You can create a toolbar out of a panel component and speed buttons, or you can use the *TToolBar* component, then right-click and choose New Button to add buttons to the toolbar.

The *TToolBar* component has several advantages: buttons on a toolbar automatically maintain uniform dimensions and spacing; other controls maintain their relative position and height; controls can automatically wrap around to start a new row when they do not fit horizontally; and *TToolBar* offers display options like transparency, pop-up borders, and spaces and dividers to group controls.

You can use a centralized set of actions on toolbars and menus, by using action lists (see page 2007) or action bands (see page 2007).

Toolbars can also parent other controls such as edit boxes, combo boxes, and so on.

See Also

TToolBar

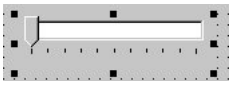
3.2.4.7.40 Track Bars

A track bar can set integer values on a continuous range. It is useful for adjusting properties like color, volume and brightness. The user moves the slide indicator by dragging it to a particular location or clicking within the bar.

- Use the *Max* and *Min* properties to set the upper and lower range of the track bar.
- Use *SelEnd* and *SelStart* to highlight a selection range.
- The *Orientation* property determines whether the track bar is vertical or horizontal.
- By default, a track bar has one row of ticks along the bottom. Use the *TickMarks* property to change their location. To control the intervals between ticks, use the *TickStyle* property and *SetTick* method.

Three views of the track bar component:







- *Position* sets a default position for the track bar and tracks the position at runtime.
- By default, users can move one tick up or down by pressing the up and down arrow keys. Set *LineSize* to change that increment.
- Set *PageSize* to determine the number of ticks moved when the user presses *Page Up* and *Page Down*.

See Also

TTrackBar

3.2.4.7.41 Tree Views

A tree view *(TTreeView*) displays items in an indented outline. The control provides buttons that allow nodes to be expanded and collapsed. You can include icons with items' text labels and display different icons to indicate whether a node is expanded or collapsed. You can also include graphics, such as check boxes, that reflect state information about the items.

- *Indent* sets the number of pixels horizontally separating items from their parents.
- *ShowButtons* enables the display of "+" and "–" buttons to indicate whether an item can be expanded.
- *ShowLines* enables display of connecting lines to show hierarchical relationships (VCL only).
- *ShowRoot* determines whether lines connecting the top-level items are displayed (VCL only).

To add items to a tree view control at design time, double-click on the control to display the TreeView Items editor. The items you add become the value of the *Items* property. You can change the items at runtime by using the methods of the *Items* property, which is an object of type *TTreeNodes*. *TTreeNodes* has methods for adding, deleting, and navigating the items in the tree view.

Tree views can display columns and subitems similar to list views in vsReport mode.

See Also

TTreeView

3.2.4.7.42 Up-down Controls (VCL Only)

In VCL applications only, an up-down control (*TUpDown*) consists of a pair of arrow buttons that allow users to change an integer value in fixed increments. The current value is given by the *Position* property; the increment, which defaults to 1, is specified by the *Increment* property. Use the *Associate* property to attach another component (such as an edit control) to the up-down control.

See Also

TUpDown

3.2.4.7.43 Display Controls

There are many ways to provide users with information about the state of an application. For example, some components—including *TForm*—have a *Caption* property that can be set at runtime. You can also create dialog boxes to display messages. In addition, the following components are especially useful for providing visual feedback at runtime to identify the object.

Use this component or property:	To do this:
TStatusBar	Display a status region (usually at the bottom of a window)
TProgressBar	Show the amount of work completed for a particular task
Hint and ShowHint	Activate fly-by or "tooltip" Help
HelpContext and HelpFile	Link context-sensitive online Help

See Also

TLabel

TProgressBar

TStatusBar

TStaticText

3.2.4.7.44 Text Controls

Many applications use text controls to display text to the user. You can use:

- Edit controls (see page 2049), which allow the user to add text.
- Text viewing controls (see page 2052) and labels (see page 2043), which do not allow user to add text.

See Also

Edit Controls (see page 2049)

Working with Text in Controls (see page 2168)

3.2.4.7.45 Text Viewing Controls

Use this component:	When you want users to do this:
TTextBrowser	Display a text file or simple HTML page that users can scroll through.

TTextViewer	Display a text file or simple HTML page. Users can scroll through the page or click links to view other pages and images.
TLCDNumber	Display numeric information in a digital display form.

TTextViewer acts as a simple viewer so that users can read and scroll through documents. With *TTextBrowser*, users can also click links to navigate to other documents and other parts of the same document. Documents visited are stored in a history list, which can be navigated using the *Backward, Forward*, and *Home* methods. *TTextViewer* and *TTextBrowser* are best used to display HTML-based text or to implement an HTML-based Help system.

TTextBrowser has the same properties as *TTextViewer* plus *Factory*. *Factory* determines the MIME factory object used to determine file types for embedded images. For example, you can associate filename extensions—such as .txt, .html, and .xml—with MIME types and have the factory load this data into the control.

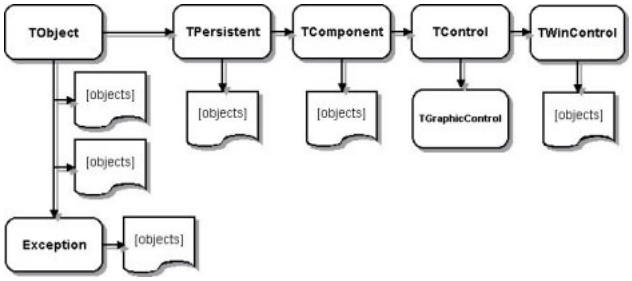
Use the *FileName* property to add a text file, such as .html, to appear in the control at runtime.

To see an application using the text browser control, see ..\Delphi7\Demos\Clx\TextBrowser.

3.2.4.8 Understanding the component library

Topics

Name	Description
Understanding the Component Library (see page 2055)	The component library is made up of objects that you can use in your projects. The VCL/RTL contains low-level classes and routines available for all VCL applications. VCL/RTL includes the runtime library (RTL) up to and including the Classes unit. Use the VCL when you want to use native Windows controls, Windows-specific features, or extend an existing VCL application. All classes in the VCL descend from *TObject*. *TObject* introduces methods that implement fundamental behavior like construction, destruction, and message handling. *Components* are a subset of the component library that descend from the class *TComponent*. You can place... more (see page 2055)
Properties, Methods, and Events (see page 2056)	The VCL hierarchy of classes is tied to the IDE, where you can develop applications quickly. The classes in the VCL are based on properties, methods, and events. Each class includes data members (properties), functions that operate on the data (methods), and a way to interact with users of the class (events). The VCL component library is written in the Delphi language, and it is based on the Windows API.
Types of Events (see page 2057)	The kinds of events that can occur can be divided into two main categories: • User events • System events • Internal events

Objects, Components, and Controls (see page 2057)	The following diagram is a greatly simplified view of the inheritance hierarchy that illustrates the relationship between objects, components, and controls.  Every object (class) inherits from *TObject*. Objects that can appear in the Form Designer inherit from *TPersistent* or *TComponent*. Controls, which appear to the user at runtime, inherit from *TControl*. There are two types of controls, graphic controls, which inherit from *TGraphicControl*, and windowed controls, which inherit from *TWinControl* or *TWidgetControl*. A control like *TCheckBox* inherits all the functionality of *TObject*, *TPersistent*, *TComponent*, *TControl*, and... more (see page 2057)
TObject Branch (see page 2058)	The *TObject* branch includes all VCL classes that descend from *TObject* but not from *TPersistent*. Much of the powerful capability of the component library is established by the methods that *TObject* introduces. *TObject* encapsulates the fundamental behavior common to all classes in the component library by introducing methods that provide: • The ability to respond when object instances are created or destroyed. • Class type and instance information on an object, and runtime type information (RTTI) about its published properties. • Support for handling messages (see page 1308) (VCL applications) . *TObject* is the immediate ancestor of many simple classes. Classes in the *TObject... more (see page 2058)*
TPersistent Branch (see page 2059)	The *TPersistent* branch includes all VCL classes that descend from *TPersistent* but not from *TComponent*. Persistence determines what gets saved with a form file or data module and what gets loaded into the form or data module when it is retrieved from memory. Because of their persistence, objects from this branch can appear at design time. However, they can't exist independently. Rather, they implement properties for components. Properties are only loaded and saved with a form if they have an owner. The owner must be some component. *TPersistent* introduces the *GetOwner* method, which lets the Form Designer determine the... more (see page 2059)
TComponent Branch (see page 2059)	The *TComponent* branch contains classes that descend from *TComponent* but not *TControl*. Objects in this branch are components that you can manipulate on forms at design time but which do not appear to the user at runtime. They are persistent objects that can do the following: • Appear on the **Tool palette** and be changed on the form. • Own and manage other components. • Load and save themselves. Several methods introduced by *TComponent* dictate how components act during design time and what information gets saved with the component. Streaming (the saving and loading of form files, which store information about the... more (see page 2059)

TControl Branch (see page 2060)	The *TControl* branch consists of components that descend from *TControl* but not *TWinControl* applications). Classes in this branch are controls: visual objects that the user can see and manipulate at runtime. All controls have properties, methods, and events in common that relate to how the control looks, such as its position, the cursor associated with the control's window, methods to paint or move the control, and events to respond to mouse actions. Controls in this branch, however, can never receive keyboard input. Whereas *TComponent* defines behavior for all components, *TControl* defines behavior for all visual controls.... more (see page 2060)
TWinControl/TWidgetControl Branch (see page 2061)	Most controls fall into the TWinControl/ *TWidgetControl* branch. Unlike graphic controls, controls in this branch have their own associated window or widget. Because of this, they are sometimes called windowed controls controls. Windowed controls all descend from *TWinControl*, which descends from the windows-only version of *TControl*. Controls in the *TWinControl*/*TWidgetControl* branch: • Can receive focus while an application is running, which means they can receive keyboard input from the application user. In comparison, graphic controls can only display data and respond to the mouse. • Can be the parent of one or more child controls. • Have a... more (see page 2061)

3.2.4.8.1 Understanding the Component Library

The component library is made up of objects that you can use in your projects. The VCL/RTL contains low-level classes and routines available for all VCL applications. VCL/RTL includes the runtime library (RTL) up to and including the Classes unit.

Use the VCL when you want to use native Windows controls, Windows-specific features, or extend an existing VCL application.

All classes in the VCL descend from *TObject*. *TObject* introduces methods that implement fundamental behavior like construction, destruction, and message handling.

Components are a subset of the component library that descend from the class *TComponent*. You can place components on a form or data module and manipulate them at design time. Using the **Object Inspector**, you can assign property values without writing code. Most components are either visual or nonvisual, depending on whether they are visible at runtime. Some components appear on the **Tool palette**.

Visual components, such as *TForm* and *TSpeedButton*, are called *controls* and descend from *TControl*. Controls are used in GUI applications, and appear to the user at runtime. *TControl* provides properties (see page 2056) that specify the visual attributes of controls, such as their height and width.

Nonvisual components are used for a variety of tasks. For example, if you are writing an application that connects to a database, you can place a *TDataSource* component on a form to connect a control and a dataset used by the control. This connection is not visible to the user, so *TDataSource* is nonvisual. At design time, nonvisual components are represented by an icon. This allows you to manipulate their properties and events just as you would a visual control.

Classes that are not components (that is, classes that descend from *TObject* but not *TComponent*) are also used for a variety of tasks. Typically, these classes are used for accessing system objects (such as a file or the clipboard) or for transient tasks (such as storing data in a list). You can't create instances of these classes at design time, although they are sometimes created by the components that you add in the Form Designer.

Detailed reference material on all VCL objects is accessible while you are programming. In the Code editor, place the cursor anywhere on the object and press F1 to display the Help topic. Objects, properties, methods, and events that are in the VCL are marked "VCL Reference."

See Also

Properties (see page 2056)

Objects (see page 2057)

3.2.4.8.2 Properties, Methods, and Events

The VCL hierarchy of classes is tied to the IDE, where you can develop applications quickly. The classes in the VCL are based on properties, methods, and events. Each class includes data members (properties), functions that operate on the data (methods), and a way to interact with users of the class (events). The VCL component library is written in the Delphi language, and it is based on the Windows API.

Properties

Properties are characteristics of an object that influence either the visible behavior or the operations of the object. For example, the *Visible* property determines whether an object can be seen in an application interface. Well-designed properties make your components easier for others to use and easier for you to maintain.

Here are some of the useful features of properties:

- Unlike methods, which are only available at runtime, you can see and change some properties at design time and get immediate feedback as the components change in the IDE.
- You can access some properties in the **Object Inspector**, where you can modify the values of your object visually. Setting properties at design time is easier than writing code and makes your code easier to maintain.
- Because the data is encapsulated, it is protected and private to the actual object.
- The calls to get and set the values of properties can be methods, so special processing can be done that is invisible to the user of the object. For example, data could reside in a table, but could appear as a normal data member to the programmer.
- You can implement logic that triggers events or modifies other data during the access of a property. For example, changing the value of one property may require you to modify another. You can change the methods created for the property.
- Properties can be virtual.
- A property is not restricted to a single object. Changing one property on one object can affect several objects. For example, setting the *Checked* property on a radio button affects all of the radio buttons in the group.

Methods

A *method* is a procedure that is always associated with a class. Methods define the behavior of an object. Class methods can access all the public (see page 2069) properties and fields of the class and are commonly referred to as member functions. Although most methods belong to an instance of a class, some methods belong instead to the class type. These are called class methods.

Events

An *event* is an action or occurrence detected by a program. Most modern applications are said to be event-driven, because they are designed to respond to events. In a program, the programmer has no way of predicting the exact sequence of actions a user will perform. For example, the user may choose a menu item, click a button, or mark some text. You can write code to handle the events in which you are interested, rather than writing code that always executes in the same restricted order.

Regardless of how an event is triggered, VCL objects look to see if you have written any code to handle that event. If you have, that code is executed; otherwise, the default event handling behavior takes place.

See Also

Types of Events (see page 2057)

Understanding the Component Library (see page 2055)

Objects (see page 2057)

3.2.4.8.3 Types of Events

The kinds of events that can occur can be divided into two main categories:

- User events
- System events
- Internal events

User events

User events are actions that the user initiates. Examples of user events are *OnClick* (the user clicked the mouse), *OnKeyPress* (the user pressed a key on the keyboard), and *OnDblClick* (the user double-clicked a mouse button).

System events

System events are events that the operating system fires for you. For example, the *OnTimer* event (which the Timer component issues whenever a predefined interval has elapsed), the *OnPaint* event (a component or window needs to be redrawn), and so on. Usually, system events are not directly initiated by a user action.

Internal events

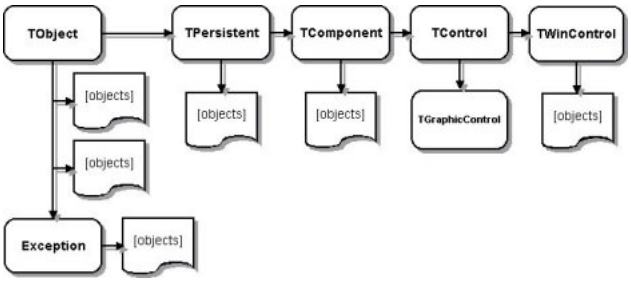
Internal events are events that are generated by the objects in your application. An example of an internal event is the *OnPost* event that a dataset generates when your application tells it to post the current record.

See Also

Properties (see page 2056)

3.2.4.8.4 Objects, Components, and Controls

The following diagram is a greatly simplified view of the inheritance hierarchy that illustrates the relationship between objects, components, and controls.



Every object (class) inherits from *TObject*. Objects that can appear in the Form Designer inherit from *TPersistent* or *TComponent*. Controls, which appear to the user at runtime, inherit from *TControl*. There are two types of controls, graphic controls, which inherit from *TGraphicControl*, and windowed controls, which inherit from *TWinControl* or *TWidgetControl*. A control like *TCheckBox* inherits all the functionality of *TObject*, *TPersistent*, *TComponent*, *TControl*, and *TWinControl* or *TWidgetControl*, and adds specialized capabilities of its own.

The figure shows several important base classes, which are described in the following table:

Class	Description
TObject	Signifies the base class and ultimate ancestor of everything in the VCL *TObject* encapsulates the fundamental behavior common to all VCLobjects by introducing methods that perform basic functions such as creating, maintaining, and destroying an instance of an object.

Exception	Specifies the base class of all classes that relate to VCL exceptions. *Exception* provides a consistent interface for error conditions, and enables applications to handle error conditions gracefully.
TPersistent	Specifies the base class for all objects that implement publishable properties. Classes under *TPersistent* deal with sending data to streams and allow for the assignment of classes.
TComponent	Specifies the base class for all components. Components can be added to the **Tool palette** and manipulated at design time. Components can own other components.
TControl	Represents the base class for all controls that are visible at runtime. *TControl* is the common ancestor of all visual components and provides standard visual controls like position and cursor. This class also provides events that respond to mouse actions.
TWinControl or TWidgetControl	Specifies the base class of all controls that can have keyboard focus. Controls under *TWinControl* are called windowed controls while those under *TWidgetControl* are called widgets.

See Also

Object Pascal and the Class Libraries (see page 2055)

TPersistent Branch (see page 2059)

TComponent Branch (see page 2059)

TControl Branch (see page 2060)

TObject Branch (see page 2058)

TWinControl/TWidgetControl Branch (see page 2061)

3.2.4.8.5 TObject Branch

The *TObject* branch includes all VCL classes that descend from *TObject* but not from *TPersistent*. Much of the powerful capability of the component library is established by the methods that *TObject* introduces. *TObject* encapsulates the fundamental behavior common to all classes in the component library by introducing methods that provide:

- The ability to respond when object instances are created or destroyed.
- Class type and instance information on an object, and runtime type information (RTTI) about its published properties.
- Support for handling messages (see page 1308) (VCL applications) .

TObject is the immediate ancestor of many simple classes. Classes in the *TObject* branch have one common, important characteristic: they are transitory. This means that these classes do not have a method to save the state that they are in prior to destruction; they are not persistent.

One of the main groups of classes in this branch is the *Exception* class. This class provides a large set of built-in exception classes for automatically handling divide-by-zero errors, file I/O errors, invalid typecasts, and many other exception conditions.

Another group in the *TObject* branch is classes that encapsulate data structures, such as:

- *TBits*, a class that stores an "array" of Boolean values.
- *TList*, a linked list class.
- *TStack*, a class that maintains a last-in first-out array of pointers.
- *TQueue*, a class that maintains a first-in first-out array of pointers.

Another group in the *TObject* branch are wrappers for external objects like *TPrinter*, which encapsulates a printer interface, and *TIniFile*, which lets a program read from or write to an ini file.

TStream is a good example of another type of class in this branch. *TStream* is the base class type for stream objects that can read from or write to various kinds of storage media, such as disk files, dynamic memory, and so on (see Using streams (see page 2099) for information on streams).

See Also

TPersistent Branch (see page 2059)

Objects (see page 2057)

TComponent Branch (see page 2059)

TControl Branch (see page 2060)

TWinControl/TWidgetControl Branch (see page 2061)

System::TObject

3.2.4.8.6 TPersistent Branch

The *TPersistent* branch includes all VCL classes that descend from *TPersistent* but not from *TComponent*. Persistence determines what gets saved with a form file or data module and what gets loaded into the form or data module when it is retrieved from memory.

Because of their persistence, objects from this branch can appear at design time. However, they can't exist independently. Rather, they implement properties for components. Properties are only loaded and saved with a form if they have an owner. The owner must be some component. *TPersistent* introduces the *GetOwner* method, which lets the Form Designer determine the owner of the object.

Classes in this branch are also the first to include a published section where properties can be automatically loaded and saved. A *DefineProperties* method lets each class indicate how to load and save properties.

Following are some of the classes in the *TPersistent* branch of the hierarchy:

- Graphics such as: *TBrush*, *TFont*, and *TPen*.
- Classes such as *TBitmap* and *TIcon*, which store and display visual images, and *TClipboard*, which contains text or graphics that have been cut or copied from an application.
- String lists, such as *TStringList*, which represent text or lists of strings that can be assigned at design time.
- Collections and collection items, which descend from *TCollection* or *TCollectionItem*. These classes maintain indexed collections of specially defined items that belong to a component. Examples include *THeaderSections* and *THeaderSection* or *TListColumns* and *TListColumn*.

See Also

Objects (see page 2057)

TComponent Branch (see page 2059)

TControl Branch (see page 2060)

TObject Branch (see page 2058)

TWinControl/TWidgetControl Branch (see page 2061)

TPersistent

3.2.4.8.7 TComponent Branch

The *TComponent* branch contains classes that descend from *TComponent* but not *TControl*. Objects in this branch are components that you can manipulate on forms at design time but which do not appear to the user at runtime. They are persistent objects that can do the following:

- Appear on the **Tool palette** and be changed on the form.

- Own and manage other components.
- Load and save themselves.

Several methods introduced by *TComponent* dictate how components act during design time and what information gets saved with the component. Streaming (the saving and loading of form files, which store information about the property values of objects on a form) is introduced in this branch. Properties are persistent if they are published and published properties are automatically streamed.

The *TComponent* branch also introduces the concept of ownership that is propagated throughout the component library. Two properties support ownership: *Owner* and *Components*. Every component has an *Owner* property that references another component as its owner. A component may own other components. In this case, all owned components are referenced in the component's *Components* property.

The constructor for every component takes a parameter that specifies the new component's owner. If the passed-in owner exists, the new component is added to that owner's Components list. Aside from using the Components list to reference owned components, this property also provides for the automatic destruction of owned components. As long as the component has an owner, it will be destroyed when the owner is destroyed. For example, since *TForm* is a descendant of *TComponent*, all components owned by a form are destroyed and their memory freed when the form is destroyed. (Assuming, of course, that the components have properly designed destructors that clean them up correctly.)

If a property type is a *TComponent* or a descendant, the streaming system creates an instance of that type when reading it in. If a property type is *TPersistent* but not *TComponent*, the streaming system uses the existing instance available through the property and reads values for that instance's properties.

Some of the classes in the *TComponent* branch include:

- *TActionList*, a class that maintains a list of actions, which provides an abstraction of the responses your program can make to user input.
- *TMainMenu*, a class that provides a menu bar and its accompanying drop-down menus for a form.
- *TOpenDialog*, TSaveDialog, TFontDialog, TFindDialog, TColorDialog, and so on, classes that display and gather information from commonly used dialog boxes.
- *TScreen*, a class that keeps track of the forms and data modules that an application creates, the active form, the active control within that form, the size and resolution of the screen, and the cursors and fonts available for the application to use.

Components that do not need a visual interface can be derived directly from *TComponent*. To make a tool such as a *TTimer* device, you can derive from *TComponent*. This type of component resides on the **Tool palette** but performs internal functions that are accessed through code rather than appearing in the user interface at runtime.

See Working with components for details on setting properties, calling methods, and working with events for components.

See Also

TPersistent Branch (see page 2059)

Objects (see page 2057)

TControl Branch (see page 2060)

TObject Branch (see page 2058)

TWinControl/TWidgetControl Branch (see page 2061)

TComponent

3.2.4.8.8 TControl Branch

The *TControl* branch consists of components that descend from *TControl* but not *TWinControl* applications*)*. Classes in this branch are controls: visual objects that the user can see and manipulate at runtime. All controls have properties, methods, and events in common that relate to how the control looks, such as its position, the cursor associated with the control's window, methods to paint or move the control, and events to respond to mouse actions. Controls in this branch, however, can never receive keyboard input.

Whereas *TComponent* defines behavior for all components, *TControl* defines behavior for all visual controls. This includes drawing routines, standard events, and containership.

TControl introduces many visual properties that all controls inherit. These include the *Caption*, *Color*, *Font*, and *HelpContext* or *HelpKeyword*. While these properties inherited from *TControl*, they are only published—and hence appear in the **Object Inspector**—for controls to which they are applicable. For example, *TImage* does not publish the *Color* property, since its color is determined by the graphic it displays. *TControl* also introduces the *Parent* property, which specifies another control that visually contains the control.

Classes in the *TControl* branch often called graphic controls, because they all descend from *TGraphicControl*, which is an immediate descendant of *TControl*. Although these controls appear to the user at runtime, graphic controls do not have their own underlying window or widget. Instead, they use their parent's window or widget. It is because of this limitation that graphic controls cant receive keyboard input or act as a parent to other controls. However, because they do not have their own window or widget, graphic controls use fewer system resources. For details on many of the classes in the *TControl* branch, see graphics controls (see page 2040).

TControl is designed for VCL (Windows-only) applications. Many controls descend from *TControl*. The Windows-only controls use native Windows APIs in their implementations.

See Also

TPersistent Branch (see page 2059)

Objects (see page 2057)

TComponent Branch (see page 2059)

TObject Branch (see page 2058)

TWinControl/TWidgetControl Branch (see page 2061)

TControl

3.2.4.8.9 TWinControl/TWidgetControl Branch

Most controls fall into the TWinControl/ *TWidgetControl* branch. Unlike graphic controls, controls in this branch have their own associated window or widget. Because of this, they are sometimes called windowed controls controls. Windowed controls all descend from *TWinControl*, which descends from the windows-only version of *TControl*.

Controls in the *TWinControl*/*TWidgetControl* branch:

- Can receive focus while an application is running, which means they can receive keyboard input from the application user. In comparison, graphic controls can only display data and respond to the mouse.
- Can be the parent of one or more child controls.
- Have a handle, or unique identifier, that allows them to access the underlying window or widget.

The *TWinControl/TWidgetControl* branch includes both controls that are drawn automatically (such as *TEdit*, *TListBox*, *TComboBox*, *TPageControl*, and so on) and custom controls that do not correspond directly to a single underlying Windows control or widget. Controls in this latter category, which includes classes like *TStringGrid* and *TDBNavigator*, must handle the details of painting themselves. Because of this, they descend from *TCustomControl*, which introduces a *Canvas* property on which they can paint themselves.

See Also

TPersistent Branch (see page 2059)

Objects (see page 2057)

TComponent Branch (see page 2059)

TControl Branch (see page 2060)

TObject Branch (see page 2058)

System::TObject

3.2.4.9 Using the object model

Topics

Name	Description
Using Object Variables (see page 2064)	You can assign one object variable to another object variable if the variables are of the same type or are assignment compatible. In particular, you can assign an object variable to another object variable if the type of the variable to which you are assigning is an ancestor of the type of the variable being assigned. For example, here is a *TSimpleForm* type declaration and a variable declaration section declaring two variables, *AForm* and *Simple*:
Changing the Name of a Component (see page 2065)	You should always use the **Object Inspector** to change the name of a component. For example, suppose you want to change a form's name from the default *Form1* to a more descriptive name, such as *ColorWindow*. When you change the form's *Name* property in the **Object Inspector**, the new name is automatically reflected in the form's .dfm or .xfm file (which you usually don't edit manually) and in the source code that the IDE generates:
Components and Ownership (see page 2066)	Delphi components have a built-in memory-management mechanism that allows one component to assume responsibility for freeing another. The former component is said to *own* the latter. The memory for an owned component is automatically freed when its owner's memory is freed. The owner of a component—the value of its *Owner* property—is determined by a parameter passed to the constructor when the component is created. By default, a form owns all components on it and is in turn owned by the application. Thus, when the application shuts down, the memory for all forms and the components on them is freed. Ownership... more (see page 2066)
Creating, Instantiating, and Destroying Objects (see page 2066)	Many of the objects you use in the Form Designer, such as buttons and edit boxes, are visible at both design time and runtime. Some, such as common dialog boxes, appear only at runtime. Still others, such as timers and data source components, have no visual representation at runtime. You may want to create your own classes. For example, you could create a *TEmployee* class that contains *Name*, *Title*, and *HourlyPayRate* properties. You could then add a *CalculatePay* method that uses the data in *HourlyPayRate* to compute a paycheck amount. The *TEmployee* type declaration might look like this:... more (see page 2066)
Examining a Delphi Object (see page 2067)	When you create a new project, the IDE displays a new form for you to customize. In the Code editor, the automatically generated unit declares a new class type for the form and includes the code that creates the new form instance. The generated code for a new Windows application looks like this:
Inheriting Data and Code from an Object (see page 2069)	The *TForm1* object in examining a Delphi object (see page 2067) seems simple. *TForm1* appears to contain one field (*Button1*), one method (*Button1Click*), and no properties. Yet you can show, hide, or resize of the form, add or delete standard border icons, and set up the form to become part of a Multiple Document Interface (MDI) application. You can do these things because the form has *inherited* all the properties and methods of the component *TForm*. When you add a new form to your project, you start with TForm and customize it by adding components, changing property values,... more (see page 2069)
Private, Protected, Public, and Published Declarations (see page 2069)	A class type declaration contains three or four possible sections that control the accessibility of its fields and methods:
Scope and Qualifiers (see page 2070)	*Scope* determines the accessibility of an object's fields, properties, and methods. All members declared in a class are available to that class and, as is discussed later, often to its descendants. Although a method's implementation code appears outside of the class declaration, the method is still within the scope of the class because it is declared in the class declaration. When you write code to implement a method that refers to properties, methods, or fields of the class where the method is declared, you don't need to preface those identifiers with the name of the class. For example, if you... more (see page 2070)

Using the Object Model (see page 2071)	The Delphi language is a set of object-oriented extensions to standard Pascal. Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once you define a class, you and other programmers can use it in different applications, thus reducing development time and increasing productivity. The following topics provide a brief introduction to object-oriented concepts for programmers who are just starting out with the Delphi language. For more details on object-oriented programming for programmers who want to write components that can be installed on the **Tool palette**, see Overview of Component Creation (see page 1313)... more (see page 2071)
What Is an Object? (see page 2071)	A *class* is a data type that encapsulates *data* and *operations on data* in a single unit. Before object-oriented programming, data and operations (functions) were treated as separate elements. An *object* is an instance of a class. That is, it is a value whose type is a class. The term object is often used more loosely in this documentation and where the distinction between a class and an instance of the class is not important, the term "object" may also refer to a class. You can begin to understand objects if you understand Pascal records or *structures* in C. Records... more (see page 2071)
TInterfacedObject (see page 2072)	When defining a class that supports one or more interfaces, it is convenient to use *TInterfacedObject* as a base class because it implements the methods of *IInterface*. *TInterfacedObject* class is declared in the *System* unit as follows:
Defining New Classes (Delphi) (see page 2072)	Although there are many classes in the object hierarchy, you are likely to need to create additional classes if you are writing object-oriented programs. The classes you write must descend from *TObject* or one of its descendants. The advantage of using classes comes from being able to create new classes as descendants of existing ones. Each descendant class inherits the fields and methods of its parent and ancestor classes. You can also declare methods in the new class that override inherited ones, introducing new, more specialized behavior. The general syntax of a descendant class is as follows:
Aggregation (see page 2074)	Aggregation offers a modular approach to code reuse through sub-objects that make up the functionality of a containing object, but that hide the implementation details from that object. In aggregation, an outer object implements one or more interfaces. At a minimum, it must implement *IInterface*. The inner object, or objects, also implement one or more interfaces. However, only the outer object exposes the interfaces. That is, the outer object exposes both the interfaces it implements and the ones that its contained objects implement. Clients know nothing about inner objects. While the outer object provides access to the inner object... more (see page 2074)
Implementing IInterface (see page 2075)	Just as all objects descend, directly or indirectly, from *TObject*, all interfaces derive from the *IInterface* interface. IInterface provides for dynamic querying and lifetime management of the interface. This is established in the three *IInterface* methods: • *QueryInterface* dynamically queries a given object to obtain interface references for the interfaces that the object supports. • *_AddRef* is a reference counting method that increments the count each time a call to QueryInterface succeeds. While the reference count is nonzero the object must remain in memory. • *_Release* is used with _AddRef to allow an object to track its own lifetime and determine when... more (see page 2075)
Memory Management of Interface Objects (see page 2075)	One of the concepts behind the design of interfaces is ensuring the lifetime management of the objects that implement them. The *_AddRef* and *_Release* methods of *IInterface* provide a way to implement this lifetime management. *_AddRef* and *_Release* track the lifetime of an object by incrementing the reference count on the object when an interface reference is passed to a client, and will destroy the object when that reference count is zero. If you are creating COM objects for distributed applications (in the Windows environment only), then you should strictly adhere to the reference counting rules. However, if you are... more (see page 2075)

Not Using Reference Counting (see page 2076)	If your object is a component or a control that is owned by another component, then it is part of a different memory management system that is based in *TComponent*. Although some classes mix the object lifetime management approaches of *TComponent* and interface reference counting, this is very tricky to implement correctly. To create a component that supports interfaces but bypasses the interface reference counting mechanism, you must implement the *_AddRef* and *_Release* methods in code such as the following:
Reusing Code and Delegation (see page 2076)	One approach to reusing code with interfaces is to have one interfaced object contain, or be contained by another. Using properties that are object types provides an approach to containment and code reuse. To support this design for interfaces, the Delphi language has a keyword implements, that makes if easy to write code to delegate (see page 2078) all or part of the implementation of an interface to a subobject. Aggregation (see page 2074) is another way of reusing code through containment and delegation. In aggregation, an outer object uses an inner object that implements interfaces which are exposed only by the outer object.
Using Interfaces Across the Hierarchy (see page 2077)	Using interfaces lets you separate the way a class is used from the way it is implemented. Two classes can implement the same interface without descending from the same base class. By obtaining an interface from either class, you can call the same methods without having to know the type of the class. This polymorphic use of the same methods on unrelated objects is possible because the objects implement the same interface. For example, consider the interface,
Using Implements for Delegation (see page 2078)	Many classes have properties that are subobjects. You can also use interfaces as property types. When a property is of an interface type (or a class type that implements the methods of an interface) you can use the keyword implements to specify that the methods of that interface are delegated to the object or interface reference which is the value of the property. The delegate only needs to provide implementation for the methods. It does not have to declare the interface support. The class containing the property must include the interface in its ancestor list. By default, using the implements... more (see page 2078)
Using Interfaces (see page 2079)	Delphi is a single-inheritance language. That means that any class has only a single direct ancestor. However, there are times you want a new class to inherit properties and methods from more than one base class so that you can use it sometimes like one and sometimes like the other. Interfaces let you achieve something like this effect. An interface is like a class that contains only abstract methods (methods with no implementation) and a clear definition of their functionality. Interface method definitions include the number and types of their parameters, their return type, and their expected behavior. By convention,... more (see page 2079)
Using Interfaces in Distributed Applications (see page 2079)	In VCL applications, interfaces are a fundamental element in the COM and SOAP distributed object models. Delphi provides base classes for these technologies that extend the basic interface functionality in *TInterfacedObject*, which simply implements the *IInterface* interface methods. When using COM, classes and interfaces are defined in terms of *IUnknown* rather than *IInterface*. There is no semantic difference between *IUnknown* and *IInterface*, the use of *IUnknown* is simply a way to adapt Delphi interfaces to the COM definition. COM classes add functionality for using class factories and class identifiers (CLSIDs). Class factories are responsible for creating class... more (see page 2079)
Using Interfaces with Procedures (see page 2080)	Interfaces allow you to write generic procedures that can handle objects without requiring that the objects descend from a particular base class. Using the *IPaint* and *IRotate* interfaces defined previously, you can write the following procedures:
Using Reference Counting (see page 2081)	The Delphi compiler provides most of the *IInterface* memory management for you by its implementation of interface querying and reference counting. Therefore, if you have an object that lives and dies by its interfaces, you can easily use reference counting by deriving from *TInterfacedObject*. If you decide to use reference counting, then you must be careful to only hold the object as an interface reference, and to be consistent in your reference counting. For example:
Using the as Operator with Interfaces (see page 2082)	Classes that implement interfaces can use the as operator for dynamic binding on the interface. In the following example,

3.2.4.9.1 Using Object Variables

You can assign one object variable to another object variable if the variables are of the same type or are assignment compatible. In particular, you can assign an object variable to another object variable if the type of the variable to which you are assigning is an ancestor of the type of the variable being assigned. For example, here is a *TSimpleForm* type declaration and a variable

declaration section declaring two variables, *AForm* and *Simple*:

```
type
  TSimpleForm = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  AForm: TForm;
  SimpleForm: TSimpleForm;
```

AForm is of type *TForm*, and *SimpleForm* is of type *TSimpleForm*. Because *TSimpleForm* is a descendant of *TForm*, this assignment statement is legal:

```
AForm := SimpleForm;
```

Suppose you write an event handler for the *OnClick* event of a button. When the button is clicked, the event handler for the *OnClick* event is called. Each event handler has a *Sender* parameter of type *TObject*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
.
.
.
end;
```

Because *Sender* is of type *TObject*, any object can be assigned to *Sender*. The value of *Sender* is always the control or component that responds to the event. You can test *Sender* to find the type of component or control that called the event handler using the reserved word is. For example,

```
if Sender is TEdit then
  DoSomething
else
  DoSomethingElse;
```

See Also

Object Pascal and the Class Libraries (see page 2055)

Scope and Qualifiers (see page 2070)

Creating (see page 2066)

3.2.4.9.2 Changing the Name of a Component

You should always use the **Object Inspector** to change the name of a component. For example, suppose you want to change a form's name from the default *Form1* to a more descriptive name, such as *ColorWindow*. When you change the form's *Name* property in the **Object Inspector**, the new name is automatically reflected in the form's .dfm or .xfm file (which you usually don't edit manually) and in the source code that the IDE generates:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;
type
  TColorWindow = class(TForm){ Changed from TForm1 to TColorWindow }
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
  end;
var
  ColorWindow: TColorWindow;{ Changed from Form1 to ColorWindow }
implementation
{$R *.dfm}
procedure TColorWindow.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;{ The reference to Form1 didn"t change! }
end;
end.
```

Note that the code in the *OnClick* event handler for the button hasn't changed. Because you wrote the code, you have to update it yourself and correct any references to the form:

```
procedure TColorWindow.Button1Click(Sender: TObject);
begin
  ColorWindow.Color := clGreen;
end;
```

See Also

Examining a Delphi Object (see page 2067)

What is an Object? (see page 2071)

3.2.4.9.3 Components and Ownership

Delphi components have a built-in memory-management mechanism that allows one component to assume responsibility for freeing another. The former component is said to *own* the latter. The memory for an owned component is automatically freed when its owner's memory is freed. The owner of a component—the value of its *Owner* property—is determined by a parameter passed to the constructor when the component is created. By default, a form owns all components on it and is in turn owned by the application. Thus, when the application shuts down, the memory for all forms and the components on them is freed.

Ownership applies only to *TComponent* and its descendants. If you create, for example, a *TStringList* or *TCollection* object (even if it is associated with a form), you are responsible for freeing the object.

See Also

Creating (see page 2066)

Memory Management of Interface Objects (see page 2075)

3.2.4.9.4 Creating, Instantiating, and Destroying Objects

Many of the objects you use in the Form Designer, such as buttons and edit boxes, are visible at both design time and runtime. Some, such as common dialog boxes, appear only at runtime. Still others, such as timers and data source components, have no visual representation at runtime.

You may want to create your own classes. For example, you could create a *TEmployee* class that contains *Name*, *Title*, and *HourlyPayRate* properties. You could then add a *CalculatePay* method that uses the data in *HourlyPayRate* to compute a paycheck amount. The *TEmployee* type declaration might look like this:

```
type
  TEmployee = class(TObject)
  private
    FName: string;
    FTitle: string;
    FHourlyPayRate: Double;
  public
    property Name: string read FName write FName;
    property Title: string read FTitle write FTitle;
    property HourlyPayRate: Double read FHourlyPayRate write FHourlyPayRate;
```

```
    function CalculatePay: Double;
  end;
```

In addition to the fields, properties, and methods you've defined, *TEmployee* inherits all the methods of *TObject*. You can place a type declaration like this one in either the interface or implementation part of a unit, and then create instances of the new class by calling the *Create* method that *TEmployee* inherits from *TObject*:

```
var
  Employee: TEmployee;
begin
  Employee := TEmployee.Create;
end;
```

The *Create* method is called a *constructor*. It allocates memory for a new instance object and returns a reference to the object.

Components on a form are created and destroyed automatically. However, if you write your own code to instantiate objects, you are responsible for disposing of them as well. Every object inherits a *Destroy* method (called a *destructor*) from *TObject*. To destroy an object, however, you should call the *Free* method (also inherited from *TObject*), because *Free* checks for a nil reference before calling *Destroy*. For example,

```
Employee.Free;
```

destroys the *Employee* object and deallocates its memory.

See Also

Object Pascal and the Class Libraries (see page 2055)

Components and Ownership (see page 2066)

Using Object Variables (see page 2064)

3.2.4.9.5 Examining a Delphi Object

When you create a new project, the IDE displays a new form for you to customize. In the Code editor, the automatically generated unit declares a new class type for the form and includes the code that creates the new form instance. The generated code for a new Windows application looks like this:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;
type
TForm1 = class(TForm){ The type declaration of the form begins here }
  private
    { Private declarations }
  public
    { Public declarations }
  end;{ The type declaration of the form ends here }
var
  Form1: TForm1;
implementation{ Beginning of implementation part }
{$R *.dfm}
end.{ End of implementation part and unit}
```

The new class type is *TForm1*, and it is derived from type *TForm*, which is also a class.

A class is like a record in that they both contain data fields, but a class also contains methods—code that acts on the object's data. So far, *TForm1* appears to contain no fields or methods, because you haven't added any components (the fields of the new object) to the form and you haven't created any event handlers (the methods of the new object). *TForm1* does contain inherited fields and methods, even though you don't see them in the type declaration.

This variable declaration declares a variable named *Form1* of the new type *TForm1*.

```
var
```

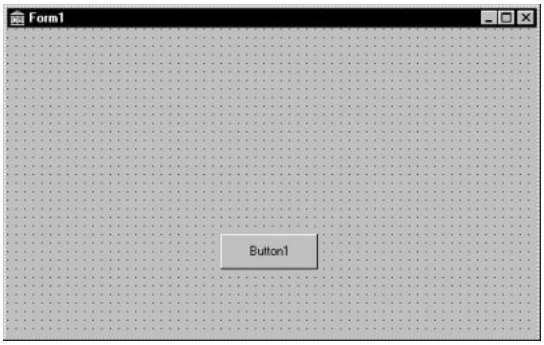
```
  Form1: TForm1;
```

Form1 represents an instance, or object, of the class type *TForm1*. You can declare more than one instance of a class type; you might want to do this, for example, to create multiple child windows in a Multiple Document Interface (MDI) application. Each instance maintains its own data, but all instances use the same code to execute methods.

Although you haven't added any components to the form or written any code, you already have a complete GUI application that you can compile and run. All it does is display a blank form.

Suppose you add a button component to this form and write an *OnClick* event handler that changes the color of the form when the user clicks the button. The result might look like this:

A simple form



When the user clicks the button, the form's color changes to green. This is the event-handler code for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Color := clGreen;
end;
```

Objects can contain other objects as data fields. Each time you place a component on a form, a new field appears in the form's type declaration. If you create the application described above and look at the code in the Code editor, this is what you see:

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
    Button1: TButton;{ New data field }
    procedure Button1Click(Sender: TObject);{ New method declaration }
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure TForm1.Button1Click(Sender: TObject);{ The code of the new method }
begin
  Form1.Color := clGreen;
end;
end.
```

TForm1 has a *Button1* field that corresponds to the button you added to the form. *TButton* is a class type, so *Button1* refers to an object.

All the event handlers you write using the IDE are methods of the form object. Each time you create an event handler, a method is declared in the form object type. The *TForm1* type now contains a new method, the *Button1Click* procedure, declared in the *TForm1* type declaration. The code that implements the *Button1Click* method appears in the implementation part of the unit.

See Also

Changing the Name of a Component (see page 2065)

What is an Object? (see page 2071)

Defining New Classes (see page 2072)

3.2.4.9.6 Inheriting Data and Code from an Object

The *TForm1* object in examining a Delphi object (see page 2067) seems simple. *TForm1* appears to contain one field (*Button1*), one method (*Button1Click*), and no properties. Yet you can show, hide, or resize of the form, add or delete standard border icons, and set up the form to become part of a Multiple Document Interface (MDI) application. You can do these things because the form has *inherited* all the properties and methods of the component *TForm*. When you add a new form to your project, you start with TForm and customize it by adding components, changing property values, and writing event handlers. To customize any object, you first derive a new object from the existing one; when you add a new form to your project, the IDE automatically derives a new form from the *TForm* type:

```
TForm1 = class(TForm)
```

A derived class inherits all the properties, events, and methods of the class from which it derives. The derived class is called a *descendant* and the class from which it derives is called an *ancestor*. If you look up *TForm* in the online Help, you'll see lists of its properties, events, and methods, including the ones that *TForm* inherits from *its* ancestors. A Delphi class can have only one immediate ancestor, but it can have many direct descendants.

See Also

What is an Object? (see page 2071)

Properties (see page 2056)

Scope and Qualifiers (see page 2070)

Defining New Classes (see page 2072)

3.2.4.9.7 Private, Protected, Public, and Published Declarations

A class type declaration contains three or four possible sections that control the accessibility of its fields and methods:

```
Type
  TClassName = Class(TObject)
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

- The public section declares fields and methods with no access restrictions. Class instances and descendant classes can access these fields and methods. A public member is accessible from wherever the class it belongs to is accessible—that is, from the unit where the class is declared and from any unit that uses that unit.
- The protected section includes fields and methods with some access restrictions. A protected member is accessible within the unit where its class is declared and by any descendant class, regardless of the descendant class's unit.
- The private section declares fields and methods that have rigorous access restrictions. A private member is accessible only within the unit where it is declared. Private members are often used in a class to implement other (public or published)

methods and properties.

- For classes that descend from *TPersistent*, a published section declares properties and events that are available at design time. A published member has the same visibility as a public member, but the compiler generates runtime type information for published members. Published properties appear in the **Object Inspector** at design time.

When you declare a field, property, or method, the new member is added to one of these four sections, which gives it its *visibility*: private, protected, public, or published.

See Also

Object Pascal and the Class Libraries (see page 2055)

Properties (see page 2056)

Scope and Qualifiers (see page 2070)

Defining New Classes (see page 2072)

3.2.4.9.8 Scope and Qualifiers

Scope determines the accessibility of an object's fields, properties, and methods. All members declared in a class are available to that class and, as is discussed later, often to its descendants. Although a method's implementation code appears outside of the class declaration, the method is still within the scope of the class because it is declared in the class declaration.

When you write code to implement a method that refers to properties, methods, or fields of the class where the method is declared, you don't need to preface those identifiers with the name of the class. For example, if you put a button on a new form, you could write this event handler for the button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Color := clFuchsia;
  Button1.Color := clLime;
end;
```

The first statement is equivalent to

```
Form1.Color := clFuchsia
```

You don't need to qualify *Color* with *Form1* because the *Button1Click* method is part of *TForm1*; identifiers in the method body therefore fall within the scope of the *TForm1* instance where the method is called. The second statement, in contrast, refers to the color of the button object (not of the form where the event handler is declared), so it requires qualification.

The IDE creates a separate unit (source code) file for each form. If you want to access one form's components from another form's unit file, you need to qualify the component names, like this:

```
Form2.Edit1.Color := clLime;
```

In the same way, you can access a component's methods from another form. For example,

```
Form2.Edit1.Clear;
```

To access *Form2*'s components from *Form1*'s unit file, you must also add *Form2*'s unit to the uses clause of *Form1*'s unit.

The scope of a class extends to its descendants. You can, however, redeclare a field, property, or method in a descendant class. Such redeclarations either hide or override the inherited member.

See Also

Object Pascal and the Class Libraries (see page 2055)

Properties (see page 2056)

Private (see page 2069)

3.2.4.9.9 Using the Object Model

The Delphi language is a set of object-oriented extensions to standard Pascal. Object-oriented programming is an extension of structured programming that emphasizes code reuse and encapsulation of data with functionality. Once you define a class, you and other programmers can use it in different applications, thus reducing development time and increasing productivity.

The following topics provide a brief introduction to object-oriented concepts for programmers who are just starting out with the Delphi language. For more details on object-oriented programming for programmers who want to write components that can be installed on the **Tool palette**, see Overview of Component Creation (see page 1313).

- What is an object? (see page 2071)
- Inheriting data and code from an object (see page 2069)
- Scope and qualifiers (see page 2070)
- Using object variables (see page 2064)
- Creating (see page 2066)
- Defining new classes (see page 2072)
- Using interfaces (see page 2079)

See Also

Object Pascal and the Class Libraries (see page 2055)

Overview of Component Creation (see page 1313)

Understanding the Class Libraries (see page 2072)

3.2.4.9.10 What Is an Object?

A *class* is a data type that encapsulates *data* and *operations on data* in a single unit. Before object-oriented programming, data and operations (functions) were treated as separate elements. An *object* is an instance of a class. That is, it is a value whose type is a class. The term object is often used more loosely in this documentation and where the distinction between a class and an instance of the class is not important, the term "object" may also refer to a class.

You can begin to understand objects if you understand Pascal records or *structures* in C. Records are made of up fields that contain data, where each field has its own type. Records make it easy to refer to a collection of varied data elements.

Objects are also collections of data elements. But objects—unlike records—contain procedures and functions that operate on their data. These procedures and functions are called *methods*.

An object's data elements are accessed through *properties*. The properties of many Delphi objects have values that you can change at design time without writing code. If you want a property value to change at runtime, you need to write only a small amount of code.

The combination of data and functionality in a single unit is called *encapsulation*. In addition to encapsulation, object-oriented programming is characterized by *inheritance* and *polymorphism*. Inheritance means that objects derive functionality from other objects (called *ancestors*); objects can modify their inherited behavior. Polymorphism means that different objects derived from the same ancestor support the same method and property interfaces, which often can be called interchangeably.

See Also

Understanding the Class Libraries (see page 2072)

Inheriting Data and Code from an Object (see page 2069)

Scope and Qualifiers (see page 2070)

Using Object Variables (see page 2064)

Creating (see page 2066)

Defining New Classes (see page 2072)

Using Interfaces (see page 2079)

3.2.4.9.11 TInterfacedObject

When defining a class that supports one or more interfaces, it is convenient to use *TInterfacedObject* as a base class because it implements the methods of *IInterface*. *TInterfacedObject* class is declared in the *System* unit as follows:

```
type
  TInterfacedObject = class(TObject, IInterface)
  protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  public
    procedure AfterConstruction; override;
procedure BeforeDestruction; override;
class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
  end;
```

Deriving directly from *TInterfacedObject* is straightforward. In the following example declaration, *TDerived* is a direct descendant of *TInterfacedObject* and implements a hypothetical *IPaint* interface.

```
type
  TDerived = class(TInterfacedObject, IPaint)
  .
  .
  .
  end;
```

Because it implements the methods of *IInterface*, *TInterfacedObject* automatically handles reference counting and memory management of interfaced objects. For more information, see Memory management of interface objects (see page 2075), which also discusses writing your own classes that implement interfaces but that do not follow the reference-counting mechanism inherent in *TInterfacedObject*.

See Also

Using Interfaces Across the Hierarchy (see page 2077)

Using Interfaces with Procedures (see page 2080)

Implementing IInterface (see page 2075)

Memory Management of Interface Objects (see page 2075)

3.2.4.9.12 Defining New Classes (Delphi)

Although there are many classes in the object hierarchy, you are likely to need to create additional classes if you are writing object-oriented programs. The classes you write must descend from *TObject* or one of its descendants.

The advantage of using classes comes from being able to create new classes as descendants of existing ones. Each descendant class inherits the fields and methods of its parent and ancestor classes. You can also declare methods in the new class that override inherited ones, introducing new, more specialized behavior.

The general syntax of a descendant class is as follows:

```
Type
  TClassName = Class (TParentClass)
    public
      {public fields}
      {public methods}
    protected
      {protected fields}
      {protected methods}
    private
      {private fields}
      {private methods}
end;
```

If no parent class name is specified, the class inherits directly from *TObject*. *TObject* defines only a handful of methods, including a basic constructor and destructor.

To define a class:

1. In the IDE, start with a project open and choose **File** ▶ **New** ▶ **Unit** to create a new unit where you can define the new class.
2. Add the uses clause and type section to the interface section.
3. In the type section, write the class declaration. You need to declare all the member variables, properties, methods, and events.

```
TMyClass = class; {This implicitly descends from TObject}
public
.
.
.
private
.
.
.
published {If descended from TPersistent or below}
.
.
.
```

If you want the class to descend from a specific class, you need to indicate that class in the definition:

```
TMyClass = class(TParentClass); {This descends from TParentClass}
```

For example:

```
type TMyButton = class(TButton)
  property Size: Integer;
  procedure DoSomething;
end;
```

4. Some editions of the IDE include a feature called class completion that simplifies the work of defining and implementing new classes by generating skeleton code for the class members you declare. If you have code completion, invoke it to finish the class declaration: place the cursor within a method definition in the interface section and press `Ctrl+Shift+C` (or right-click and select Complete Class at Cursor). Any unfinished property declarations are completed, and for any methods that require an implementation, empty methods are added to the implementation section. If you do not have class completion, you need to write the code yourself, completing property declarations and writing the methods. Given the example above, if you have class completion, read and write specifiers are added to your declaration, including any supporting fields or methods:

```
type TMyButton = class(TButton)
  property Size: Integer read FSize write SetSize;
  procedure DoSomething;
private
  FSize: Integer;
  procedure SetSize(const Value: Integer);
```

The following code is also added to the implementation section of the unit.

```
{ TMyButton }
procedure TMyButton.DoSomething;
```

```
begin
end;
procedure TMyButton.SetSize(const Value: Integer);
begin
FSize := Value;
end;
```

5. Fill in the methods. For example, to make it so the button beeps when you call the *DoSomething* method, add the Beep between begin and end.

```
{ TMyButton }
procedure TMyButton.DoSomething;
begin
  Beep;
end;
procedure TMyButton.SetSize(const Value: Integer);
begin
  if fsize < > value then
  begin
    FSize := Value;
    DoSomething;
  end;
end;
```

Note that the button also beeps when you call *SetSize* to change the size of the button.

See Also

Inheriting Data and Code from an Object (see page 2069)

Scope and Qualifiers (see page 2070)

Using Object Variables (see page 2064)

Creating (see page 2066)

Using Interfaces (see page 2079)

3.2.4.9.13 Aggregation

Aggregation offers a modular approach to code reuse through sub-objects that make up the functionality of a containing object, but that hide the implementation details from that object. In aggregation, an outer object implements one or more interfaces. At a minimum, it must implement *IInterface*. The inner object, or objects, also implement one or more interfaces. However, only the outer object exposes the interfaces. That is, the outer object exposes both the interfaces it implements and the ones that its contained objects implement.

Clients know nothing about inner objects. While the outer object provides access to the inner object interfaces, their implementation is completely transparent. Therefore, the outer object class can exchange the inner object class type for any class that implements the same interface. Correspondingly, the code for the inner object classes can be shared by other classes that want to use it.

The aggregation model defines explicit rules for implementing *IInterface* using delegation. The inner object must implement two versions of the *IInterface* methods.

- It must implement *IInterface* on itself, controlling its own reference count. This implementation of *IInterface* tracks the relationship between the outer and the inner object. For example, when an object of its type (the inner object) is created, the creation succeeds only for a requested interface of type *IInterface*.
- It also implements a second *IInterface* for all the interfaces it implements that the outer object exposes. This second *IInterface* delegates calls to *QueryInterface*, *_AddRef*, and *_Release* to the outer object. The outer *IInterface* is referred to as the "controlling Unknown."

Refer to the MS online help for the rules about creating an aggregation. When writing your own aggregation classes, you can also refer to the implementation details of *IInterface* in *TComObject*. *TComObject* is a COM class that supports aggregation. If

you are writing COM applications, you can also use *TComObject* directly as a base class.

See Also

Using Implements for Delegation (see page 2078)

Implementing IInterface (see page 2075)

3.2.4.9.14 Implementing IInterface

Just as all objects descend, directly or indirectly, from *TObject*, all interfaces derive from the *IInterface* interface. IInterface provides for dynamic querying and lifetime management of the interface. This is established in the three *IInterface* methods:

- *QueryInterface* dynamically queries a given object to obtain interface references for the interfaces that the object supports.
- *_AddRef* is a reference counting method that increments the count each time a call to QueryInterface succeeds. While the reference count is nonzero the object must remain in memory.
- *_Release* is used with _AddRef to allow an object to track its own lifetime and determine when it is safe to delete itself. Once the reference count reaches zero, the object is freed from memory. Every class that implements interfaces must implement the three IInterface methods, as well as all of the methods declared by any other ancestor interfaces, and all of the methods declared by the interface itself. You can, however, inherit the implementations of methods of interfaces declared in your class.

By implementing these methods yourself, you can provide an alternative means of lifetime management, disabling reference-counting. This is a powerful technique that lets you decouple interfaces from reference-counting.

See Also

Using Interfaces Across the Hierarchy (see page 2077)

Using Interfaces with Procedures (see page 2080)

TInterfacedObject (see page 2072)

Using the as Operator with Interfaces (see page 2082)

Reusing Code and Delegation (see page 2076)

Memory Management of Interface Objects (see page 2075)

Using Interfaces in Distributed Applications (see page 2079)

3.2.4.9.15 Memory Management of Interface Objects

One of the concepts behind the design of interfaces is ensuring the lifetime management of the objects that implement them. The *_AddRef* and *_Release* methods of *IInterface* provide a way to implement this lifetime management. *_AddRef* and *_Release* track the lifetime of an object by incrementing the reference count on the object when an interface reference is passed to a client, and will destroy the object when that reference count is zero.

If you are creating COM objects for distributed applications (in the Windows environment only), then you should strictly adhere to the reference counting rules. However, if you are using interfaces only internally in your application, then you have a choice that depends upon the nature of your object and how you decide to use it.

See Also

Using Interfaces with Procedures (see page 2080)

Implementing IInterface (see page 2075)

Using the as Operator with Interfaces (see page 2082)

Reusing Code and Delegation (see page 2076)

Using Interfaces Across the Hierarchy (see page 2077)

Using Interfaces in Distributed Applications (see page 2079)

3.2.4.9.16 Not Using Reference Counting

If your object is a component or a control that is owned by another component, then it is part of a different memory management system that is based in *TComponent*. Although some classes mix the object lifetime management approaches of *TComponent* and interface reference counting, this is very tricky to implement correctly.

To create a component that supports interfaces but bypasses the interface reference counting mechanism, you must implement the *_AddRef* and *_Release* methods in code such as the following:

```
function TMyObject._AddRef: Integer;
begin
  Result := -1;
end;
function TMyObject._Release: Integer;
begin
  Result := -1;
end;
```

You would still implement *QueryInterface* as usual to provide dynamic querying on your object.

Note that, because you implement *QueryInterface*, you can still use the as operator for interfaces, as long as you create an interface identifier (IID). You can also use aggregation. If the outer object is a component, the inner object implements reference counting as usual, by delegating to the "controlling Unknown." It is at the level of the outer object that the decision is made to circumvent the *_AddRef* and *_Release* methods, and to handle memory management via another approach. In fact, you can use *TInterfacedObject* as a base class for an inner object of an aggregation that has a as its containing outer object one that does not follow the interface lifetime model.

Note: The "controlling Unknown" is the *IUnknown* implemented by the outer object and the one for which the reference count of the entire object is maintained. *IUnknown* is the same as *IInterface*, but is used instead in COM-based applications (Windows only). For more information distinguishing the various implementations of the *IUnknown* or *IInterface* interface by the inner and outer objects, see Aggregation (see page 2074) and the Microsoft online Help topics on the "controlling Unknown."

See Also

Using Reference Counting (see page 2081)

3.2.4.9.17 Reusing Code and Delegation

One approach to reusing code with interfaces is to have one interfaced object contain, or be contained by another. Using properties that are object types provides an approach to containment and code reuse. To support this design for interfaces, the Delphi language has a keyword implements, that makes if easy to write code to delegate (see page 2078) all or part of the implementation of an interface to a subobject.

Aggregation (see page 2074) is another way of reusing code through containment and delegation. In aggregation, an outer object uses an inner object that implements interfaces which are exposed only by the outer object.

See Also

Using Interfaces with Procedures (see page 2080)

Implementing IInterface (see page 2075)

Using the as Operator with Interfaces (see page 2082)

Using Interfaces Across the Hierarchy (see page 2077)

Memory Management of Interface Objects (see page 2075)

Using Interfaces in Distributed Applications (see page 2079)

3.2.4.9.18 Using Interfaces Across the Hierarchy

Using interfaces lets you separate the way a class is used from the way it is implemented. Two classes can implement the same interface without descending from the same base class. By obtaining an interface from either class, you can call the same methods without having to know the type of the class. This polymorphic use of the same methods on unrelated objects is possible because the objects implement the same interface. For example, consider the interface,

```
IPaint = interface
  procedure Paint;
end;
```

and the two classes,

```
TSquare = class(TPolygonObject, IPaint)
  procedure Paint;
end;
TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Whether or not the two classes share a common ancestor, they are still assignment compatible with a variable of *IPaint* as in

```
var
  Painter: IPaint;
begin
  Painter := TSquare.Create;
  Painter.Paint;
  Painter := TCircle.Create;
  Painter.Paint;
end;
```

This could have been accomplished by having *TCircle* and *TSquare* descend from a common ancestor (say, *TFigure*), which declares a virtual method *Paint*. Both *TCircle* and *TSquare* would then have overridden the *Paint* method. In the previous example, *IPaint* could be replaced by *TFigure*. However, consider the following interface:

```
IRotate = interface
  procedure Rotate(Degrees: Integer);
end;
```

IRotate makes sense for the rectangle but not the circle. The classes would look like

```
TSquare = class(TRectangularObject, IPaint, IRotate)
  procedure Paint;
  procedure Rotate(Degrees: Integer);
end;
TCircle = class(TCustomShape, IPaint)
  procedure Paint;
end;
```

Later, you could create a class *TFilledCircle* that implements the *IRotate* interface to allow rotation of a pattern that fills the circle without having to add rotation to the simple circle.

Note: For these examples, the immediate base class or an ancestor class is assumed to have implemented the methods of *IInterface*, the base interface from which all interfaces descend. For more information on *IInterface*, see Implementing IInterface (see page 2075) and Memory management of interface objects (see page 2075).

See Also

Using Interfaces with Procedures (see page 2080)

Using the As Operator with Interfaces (see page 2082)

Reusing Code and Delegation (see page 2076)

Memory Management of Interface Objects (see page 2075)

Using Interfaces in Distributed Applications (see page 2079)

3.2.4.9.19 Using Implements for Delegation

Many classes have properties that are subobjects. You can also use interfaces as property types. When a property is of an interface type (or a class type that implements the methods of an interface) you can use the keyword implements to specify that the methods of that interface are delegated to the object or interface reference which is the value of the property. The delegate only needs to provide implementation for the methods. It does not have to declare the interface support. The class containing the property must include the interface in its ancestor list.

By default, using the implements keyword delegates all interface methods. However, you can use methods resolution clauses or declare methods in your class that implement some of the interface methods to override this default behavior.

The following example uses the implements keyword in the design of a color adapter object that converts an 8-bit RGB color value to a *Color* reference:

```
unit cadapt;
interface
type
IRGB8bit = interface
['{1d76360a-f4f5-11d1-87d4-00c04fb17199}']
    function Red: Byte;
    function Green: Byte;
    function Blue: Byte;
  end;
IColorRef = interface
['{1d76360b-f4f5-11d1-87d4-00c04fb17199}']
    function Color: Integer;
  end;
{ TRGB8ColorRefAdapter   map an IRGB8bit to an IColorRef }
TRGB8ColorRefAdapter = class(TInterfacedObject, IRGB8bit, IColorRef)
  private
    FRGB8bit: IRGB8bit;
    FPalRelative: Boolean;
  public
    constructor Create(rgb: IRGB8bit);
    property RGB8Intf: IRGB8bit read FRGB8bit implements IRGB8bit;
    property PalRelative: Boolean read FPalRelative write FPalRelative;
    function Color: Integer;
  end;
implementation
constructor TRGB8ColorRefAdapter.Create(rgb: IRGB8bit);
begin
  FRGB8bit := rgb;
end;
function TRGB8ColorRefAdapter.Color: Integer;
begin
  if FPalRelative then
    Result := PaletteRGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue)
  else
    Result := RGB(RGB8Intf.Red, RGB8Intf.Green, RGB8Intf.Blue);
end;
end.
```

See Also

Aggregation (see page 2074)

Implementing IInterface (see page 2075)

3.2.4.9.20 Using Interfaces

Delphi is a single-inheritance language. That means that any class has only a single direct ancestor. However, there are times you want a new class to inherit properties and methods from more than one base class so that you can use it sometimes like one and sometimes like the other. Interfaces let you achieve something like this effect.

An interface is like a class that contains only abstract methods (methods with no implementation) and a clear definition of their functionality. Interface method definitions include the number and types of their parameters, their return type, and their expected behavior. By convention, interfaces are named according to their behavior and prefaced with a capital *I*. For example, an *IMalloc* interface would allocate, free, and manage memory. Similarly, an *IPersist* interface could be used as a general base interface for descendants, each of which defines specific method prototypes for loading and saving the state of an object to a storage, stream, or file.

An interface has the following syntax:

```
IMyObject = interface
  procedure MyProcedure;
end;
```

A simple example of an interface declaration is:

```
type
IEdit = interface
  procedure Copy;
  procedure Cut;
  procedure Paste;
  function Undo: Boolean;
end;
```

Interfaces can never be instantiated. To use an interface, you need to obtain it from an implementing class.

To implement an interface, define a class that declares the interface in its ancestor list, indicating that it will implement all of the methods of that interface:

```
TEditor = class(TInterfacedObject, IEdit)
  procedure Copy;
  procedure Cut;
  procedure Paste;
  function Undo: Boolean;
end;
```

While interfaces define the behavior and signature of their methods, they do not define the implementations. As long as the class's implementation conforms to the interface definition, the interface is fully polymorphic, meaning that accessing and using the interface is the same for any implementation of it.

See Also

What is an Object? (see page 2071)

3.2.4.9.21 Using Interfaces in Distributed Applications

In VCL applications, interfaces are a fundamental element in the COM and SOAP distributed object models. Delphi provides base classes for these technologies that extend the basic interface functionality in *TInterfacedObject*, which simply implements the *IInterface* interface methods.

When using COM, classes and interfaces are defined in terms of *IUnknown* rather than *IInterface*. There is no semantic difference between *IUnknown* and *IInterface*, the use of *IUnknown* is simply a way to adapt Delphi interfaces to the COM definition. COM classes add functionality for using class factories and class identifiers (CLSIDs). Class factories are responsible for creating class instances via CLSIDs. The CLSIDs are used to register and manipulate COM classes. COM classes that have class factories and class identifiers are called CoClasses. CoClasses take advantage of the versioning capabilities of

QueryInterface, so that when a software module is updated *QueryInterface* can be invoked at runtime to query the current capabilities of an object.

New versions of old interfaces, as well as any new interfaces or features of an object, can become immediately available to new clients. At the same time, objects retain complete compatibility with existing client code; no recompilation is necessary because interface implementations are hidden (while the methods and parameters remain constant). In COM applications, developers can change the implementation to improve performance, or for any internal reason, without breaking any client code that relies on that interface. For more information about COM interfaces, see Overview of COM technologies (see page 1385).

When distributing an application using SOAP, interfaces are required to carry their own runtime type information (RTTI). The compiler only adds RTTI to an interface when it is compiled using the {$M+} switch. Such interfaces are called *invokable interfaces*. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that is not invokable, client applications can only call the methods defined in the invokable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can't be called by clients.

The easiest way to define invokable interfaces is to define your interface so that it descends from *IInvokable*. *IInvokable* is the same as *IInterface*, except that it is compiled using the {$M+} switch. For more information about Web Service applications that are distributed using SOAP, and about invokable interfaces, see Using Web Services (see page 2291).

See Also

Using Interfaces with Procedures (see page 2080)

Implementing IInterface (see page 2075)

Using the As Operator with Interfaces (see page 2082)

Reusing Code and Delegation (see page 2076)

Memory Management of Interface Objects (see page 2075)

Using Interfaces Across the Hierarchy (see page 2077)

3.2.4.9.22 Using Interfaces with Procedures

Interfaces allow you to write generic procedures that can handle objects without requiring that the objects descend from a particular base class. Using the *IPaint* and *IRotate* interfaces defined previously, you can write the following procedures:

```
procedure PaintObjects(Painters: array of IPaint);
var
  I: Integer;
begin
  for I := Low(Painters) to High(Painters) do
    Painters[I].Paint;
end;
procedure RotateObjects(Degrees: Integer; Rotaters: array of IRotate);
var
  I: Integer;
begin
  for I := Low(Rotaters) to High(Rotaters) do
    Rotaters[I].Rotate(Degrees);
end;
```

RotateObjects does not require that the objects know how to paint themselves and *PaintObjects* does not require the objects know how to rotate. This allows the generic procedures to be used more often than if they were written to only work against a *TFigure* class.

See Also

Using Interfaces Across the Hierarchy (see page 2077)

Implementing IInterface (see page 2075)

Using the As Operator with Interfaces (see page 2082)

Reusing Code and Delegation (see page 2076)

Memory Management of Interface Objects (see page 2075)

Using Interfaces in Distributed Applications (see page 2079)

3.2.4.9.23 Using Reference Counting

The Delphi compiler provides most of the *IInterface* memory management for you by its implementation of interface querying and reference counting. Therefore, if you have an object that lives and dies by its interfaces, you can easily use reference counting by deriving from *TInterfacedObject*. If you decide to use reference counting, then you must be careful to only hold the object as an interface reference, and to be consistent in your reference counting. For example:

```
procedure beep(x: ITest);
function test_func()
var
  y: ITest;
begin
  y := TTest.Create; // because y is of type ITest, the reference count is one
  beep(y); // the act of calling the beep function increments the reference count
  // and then decrements it when it returns
  y.something; // object is still here with a reference count of one
end;
```

This is the cleanest and safest approach to memory management; and if you use *TInterfacedObject* it is handled automatically. If you do not follow this rule, your object can unexpectedly disappear, as demonstrated in the following code:

```
function test_func()
var
  x: TTest;
begin
  x := TTest.Create; // no count on the object yet
  beep(x as ITest); // count is incremented by the act of calling beep
  // and decremented when it returns
  x.something; // surprise, the object is gone
end;
```

Note: In the examples above, the *beep* procedure, as it is declared, increments the reference count (call *_AddRef*) on the parameter, whereas either of the following declarations do not:

```
procedure beep(const x: ITest);
```

or

```
procedure beep(var x: ITest);
```

These declarations generate smaller, faster code.

One case where you cannot use reference counting, because it cannot be consistently applied, is if your object is a component or a control owned by another component. In that case, you can still use interfaces, but you should not use reference counting (see page 2076) because the lifetime of the object is not dictated by its interfaces.

See Also

Not Using Reference Counting (see page 2076)

3.2.4.9.24 Using the as Operator with Interfaces

Classes that implement interfaces can use the as operator for dynamic binding on the interface. In the following example,

```
procedure PaintObjects(P: TInterfacedObject)
var
  X: IPaint;
begin
  X := P as IPaint;
{ statements }
end;
```

the variable *P* of type *TInterfacedObject*, can be assigned to the variable *X,* which is an *IPaint* interface reference. Dynamic binding makes this assignment possible. For this assignment, the compiler generates code to call the *QueryInterface* method of *P*'s *IInterface* interface. This is because the compiler cannot tell from *P*'s declared type whether *P*'s instance actually supports *IPaint*. At runtime, *P* either resolves to an *IPaint* reference or an exception is raised. In either case, assigning *P* to *X* will not generate a compile-time error as it would if *P* was of a class type that did not implement *IInterface*.

When you use the as operator for dynamic binding on an interface, you should be aware of the following requirements:

- Explicitly declaring *IInterface*: Although all interfaces derive from *IInterface*, it is not sufficient, if you want to use the as operator, for a class to simply implement the methods of *IInterface*. This is true even if it also implements the interfaces it explicitly declares. The class must explicitly declare *IInterface* in its interface list.
- Using an IID: Interfaces can use an identifier that is based on a GUID (globally unique identifier). GUIDs that are used to identify interfaces are referred to as interface identifiers (IIDs). If you are using the as operator with an interface, it must have an associated IID. To create a new GUID in your source code you can use the `Ctrl+Shift+G` editor shortcut key.

See Also

Using Interfaces with Procedures (see page 2080)

Implementing IInterface (see page 2075)

Using Interfaces Across the Hierarchy (see page 2077)

Memory Management of Interface Objects (see page 2075)

3.2.4.10 Using the VCL/RTL

Topics

Name	Description
Using the VCL/RTL: Overview (see page 2090)	There are a number of units in the component library that provide the underlying support for most of the component libraries. These units include the global routines that make up the runtime library, a number of utility classes such as those that represent streams and lists, and the classes *TObject*, *TPersistent*, and *TComponent*. Collectively, these units are called the VCL/RTL. The VCL/RTL does not include any of the components that appear on the **Tool Palette**. Rather, the classes and routines in the VCL/RTL are used by the components that do appear on the **Tool Palette** and... more (see page 2090)
Adding New Measurement Types (see page 2091)	If you want to perform conversions between measurement units not already defined in the StdConvs unit, you need to create a new conversion family to represent the measurement units (*TConvType* values). When two *TConvType* values are registered with the same conversion family, the *Convert* function can convert between measurements made using the units represented by those *TConvType* values. You first need to obtain *TConvFamily* values by registering a conversion family using the RegisterConversionFamily function. After you get a *TConvFamily* value (by registering a new conversion family or using one of the global variables in the StdConvs unit),... more (see page 2091)

Approaches to File I/O (see page 2091)	There are several approaches you can take when reading from and writing to files: • The recommended approach for working with files is to use file streams. File streams are instances of the *TFileStream* class used to access information in disk files. File streams are a portable and high-level approach to file I/O. Because file streams make the file handle available, this approach can be combined with the next one. The Using file streams (see page 2133) discusses *TFileStream* in detail. • You can work with files using a handle-based approach. File handles are provided by the operating system when you create or open a... more (see page 2091)
Common List Operations (see page 2091)	Although the various list classes contain different types of items and have different ancestries, most of them share a common set of methods for adding, deleting, rearranging, and accessing the items in the list.
Converting Measurements (see page 2092)	The ConvUtils unit declares a general-purpose Conversion Function (see page 2113) that you can use to convert a measurement from one set of units to another. You can perform conversions between compatible units of measurement such as feet and inches or days and weeks. Units that measure the same types of things are said to be in the same *conversion family*. The units you're converting must be in the same conversion family. For information on doing conversions, see Performing Conversions (see page 2105). The StdConvs unit defines several conversion families and measurement units within each family. In addition, you can create customized... more (see page 2092)
Copying and Clearing Custom Variants (see page 2093)	In addition to typecasting and the implementation of operators, you must indicate how to copy and clear variants of your custom Variant type. To indicate how to copy the variant's value, implement the Copy method. Typically, this is an easy operation, although you must remember to allocate memory for any classes or structures you use to hold the variant's value:
Copying Data from One Stream to Another (see page 2094)	When copying data from one stream to another, you do not need to explicitly read and then write the data. Instead, you can use the *CopyFrom* method, as illustrated in the following example. In the following example, one file is copied to another one using streams. The application includes two edit controls (*EdFrom* and *EdTo*) and a *Copy File* button.
Creating a Class to Enable the Custom Variant Type (see page 2094)	Custom variants work by using a special helper class that indicates how variants of the custom type can perform standard operations. You create this helper class by writing a descendant of TCustomVariantType. This involves overriding the appropriate virtual methods of *TCustomVariantType*. The following topics provide details on how to implement and use a TCustomVariantType descendant: • Enabling casting (see page 2097) • Implementing binary operations (see page 2100) • Implementing comparison operations (see page 2102) • Implementing unary operations (see page 2103) • Copying and clearing custom variants (see page 2093) • Loading and saving custom variant values (see page 2104) • Using the TCustomVariantType descendant (see page 2116)
Creating a Simple Conversion Family and Adding Units (see page 2095)	One example of when you could create a new conversion family and add new measurement types might be when performing conversions between long periods of time (such as months to centuries) where a loss of precision can occur. To explain this further, the *cbTime* family uses a day as its base unit. The base unit is the one that is used when performing all conversions within that family. Therefore, all conversions must be done in terms of days. An inaccuracy can occur when performing conversions using units of months or larger (months, years, decades, centuries, millennia) because there is not... more (see page 2095)

Defining Custom Variants (see page 2096)	One powerful built-in type of the Delphi language is the Variant type. Variants represent values whose type is not determined at compile time. Instead, the type of their value can change at runtime. Variants can mix with other variants and with integer, real, string, and boolean values in expressions and assignments; the compiler automatically performs type conversions. By default, variants can't hold values that are records, sets, static arrays, files, classes, class references, or pointers. You can, however, extend the Variant type to work with any particular example of these types. All you need to do is create... more (see page 2096)
Enabling Casting (see page 2097)	One of the most important features of the custom variant type for you to implement is typecasting. The flexibility of variants arises, in part, from their implicit typecasts. There are two methods for you to implement that enable the custom Variant type to perform typecasts: Cast, which converts another variant type to your custom variant, and CastTo, which converts your custom Variant type to another type of Variant. When implementing either of these methods, it is relatively easy to perform the logical conversions from the built-in variant types. You must consider, however, the possibility that the variant to or from... more (see page 2097)
Use __fastcall When Overriding VCL Methods (see page 2098)	Member functions for VCL classes generally use __fastcall in their declaration unless declared otherwise. This means that you must use __fastcall in the declaration of any VCL class member function that you override.
Working with ini Files and the System Registry (see page 2099)	Many applications use ini files to store configuration information. The VCL/RTL includes two classes for working with ini files: *TIniFile* and *TMemIniFile*. Using ini files has the advantage that they can be used in cross-platform applications and they are easy to read and edit. For information on these classes, see Using TIniFile and TMemIniFile (see page 2131) for more information. Many Windows applications replace the use of ini files with the system Registry. The Windows system Registry is a hierarchical database that acts as a centralized storage space for configuration information. The VCL includes classes for working with the... more (see page 2099)
Using Streams (see page 2099)	Streams are classes that let you read and write data. They provide a common interface for reading and writing to different media such as memory, strings, sockets, and BLOB fields in databases. There are several stream classes, which all descend from *TStream*. Each stream class is specific to one media type. For example, *TMemoryStream* reads from or writes to a memory image; *TFileStream* reads from or writes to a file. The following topics describe the methods common to all stream classes: • Using streams to read or write data (see page 2115) • Copying data from one stream to another (see page 2094) • Specifying the stream position (see page 2106)... more (see page 2099)
Using TRegistry (see page 2099)	If you are writing a Windows-only application and are comfortable with the structure of the system Registry, you can use *TRegistry*. Unlike *TRegistryIniFile*, which uses the same properties and methods of other ini file components, the properties and methods of *TRegistry* correspond more directly to the structure of the system Registry. You can specify both the root key and subkey using *TRegistry*, while *TRegistryIniFile* uses `HKEY_CURRENT_USER` as the root key. In addition to methods for opening, closing, saving, moving, copying, and deleting keys, *TRegistry* lets you specify the access level you want to... more (see page 2099)
Implementing Binary Operations (see page 2100)	To allow the custom variant type to work with standard binary operators (+, -, *, /, div, mod, shl, shr, and, or, xor listed in the System unit), you must override the BinaryOp method. *BinaryOp* has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the operator. Implement this method to perform the operation and return the result using the same variable that contained the left-hand operand. For example, the following *BinaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:
Implementing Comparison Operations (see page 2102)	There are two ways to enable a custom variant type to support comparison operators (=, <>, <, <=, >, >=). You can override the *Compare* method, or you can override the *CompareOp* method. The Compare method is easiest if your custom variant type supports the full range of comparison operators. *Compare* takes three parameters: the left-hand operand, the right-hand operand, and a var Parameter that returns the relationship between the two. For example, the *TConvertVariantType* object in the VarConv unit implements the following *Compare* method:

Implementing Unary Operations (see page 2103)	To allow the custom variant type to work with standard unary operators (-, not), you must override the UnaryOp method. *UnaryOp* has two parameters: the value of the operand and the operator. Implement this method to perform the operation and return the result using the same variable that contained the operand. For example, the following *UnaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:
Loading and Saving Custom Variant Values (see page 2104)	By default, when the custom variant is assigned as the value of a published property, it is typecast to a string when that property is saved to a form file, and converted back from a string when the property is read from a form file. You can, however, provide your own mechanism for loading and saving custom variant values in a more natural representation. To do so, the *TCustomVariantType* descendant must implement the IVarStreamable interface from Classes.pas. *IVarStreamable* defines two methods, *StreamIn* and *StreamOut*, for reading a variant's value from a stream and for writing the variant's value to... more (see page 2104)
Performing Conversions (see page 2105)	You can use the *Convert* function to perform both simple and complex conversions. It includes a simple syntax and a second syntax for performing conversions between complex measurement types.
Persistent Lists (see page 2105)	Persistent lists can be saved to a form file. Because of this, they are often used as the type of a published property on a component. You can add items to the list at design time, and those items are saved with the object so that they are there when the component that uses them is loaded into memory at runtime. There are two main types of persistent lists: string lists and collections. Examples of string lists include *TStringList* and *THashedStringList*. String lists, as the name implies, contain strings. They provide special support for strings of the form Name=Value,... more (see page 2105)
Specifying the Stream Position and Size (see page 2106)	In addition to methods for reading and writing, streams permit applications to seek to an arbitrary position in the stream or change the size of the stream. Once you seek to a specified position, the next read or write operation starts reading from or writing to the stream at that position.
Storing a Custom Variant Type's Data (see page 2107)	Variants store their data in the *TVarData* record type. This type is a record that contains 16 bytes. The first word indicates the type of the variant, and the remaining 14 bytes are available to store the data. While your new Variant type can work directly with a *TVarData* record, it is usually easier to define a record type whose members have names that are meaningful for your new type, and cast that new type onto the *TVarData* record type. For example, the VarConv unit defines a custom variant type that represents a measurement. The data for this... more (see page 2107)
String Dependencies (see page 2108)	Sometimes you need to convert a long string to a null-terminated string, for example, if you are using a function that takes a *PChar*. If you must cast a string to a *PChar*, be aware that you are responsible for the lifetime of the resulting *PChar*. Because long strings are reference counted, typecasting a string to a *PChar* increases the dependency on the string by one, without actually incrementing the reference count. When the reference count hits zero, the string will be destroyed, even though there is an extra dependency on it. The cast *PChar* will also... more (see page 2108)
Supporting Properties and Methods in Custom Variants (see page 2108)	Some variants have properties and methods. For example, when the value of a variant is an interface, you can use the variant to read or write the values of properties on that interface and call its methods. Even if your custom variant type does not represent an interface, you may want to give it properties and methods that an application can use in the same way.
Using a Class to Manage Conversions (see page 2110)	You can always use conversion functions to register a conversion unit. There are times, however, when this requires you to create an unnecessarily large number of functions that all do essentially the same thing. If you can write a set of conversion functions that differ only in the value of a parameter or variable, you can create a class to handle those conversions. For example, there is a set standard techniques for converting between the various European currencies since the introduction of the Euro. Even though the conversion factors remain constant (unlike the conversion factor between, say, dollars and Euros),... more (see page 2110)
Using a Conversion Function (see page 2113)	For cases when the conversion is more complex, you can use a different syntax to specify a function to perform the conversion instead of using a conversion factor. For example, you can't convert temperature values using a conversion factor, because different temperature scales have a different origins. This example, which comes from the StdConvs unit, shows how to register a conversion type by providing functions to convert to and from the base units.

Using Streams to Read or Write Data (see page 2115)	Stream classes all share several methods for reading and writing data. These methods are distinguished by whether they: • Return the number of bytes read or written. • Require the number of bytes to be known. • Raise an exception on error.
Using the TCustomVariantType Descendant (see page 2116)	In the initialization section of the unit that defines your *TCustomVariantType* descendant, create an instance of your class. When you instantiate your object, it automatically registers itself with the variant-handling system so that the new Variant type is enabled. For example, here is the initialization section of the VarCmplx unit:
Writing Utilities to Work with a Custom Variant Type (see page 2117)	Once you have created a *TCustomVariantType* descendant to implement your custom variant type, it is possible to use the new Variant type in applications. However, without a few utilities, this is not as easy as it should be. It is a good idea to create a method that creates an instance of your custom variant type from an appropriate value or set of values. This function or set of functions fills out the structure you defined (see page 2107) to store your custom variant's data. For example, the following function could be used to create a complex-valued variant:
Commonly Used Long String Routines (see page 2118)	The string handling routines in `SysUtils` handle both **UnicodeString** and **AnsiString** types. Within their functional areas, some routines are used for the same purpose, the differences being whether they use a particular criterion in their calculations. The `AnsiStrings` unit's functions offer the same capabilities as the `SysUtils` unit's routines. `AnsiStrings` functions provide better performance for **AnsiString** than `SysUtils` functions, because `SysUtils` functions convert an **AnsiString** to a **UnicodeString**. `AnsiStrings` functions work only with **AnsiString**, so there is no conversion. The following tables list these routines by these functional areas: • Comparison • Case... more (see page 2118)
Commonly Used Routines for Null-terminated Strings (see page 2121)	The null-terminated string handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether or not they use a particular criteria in their calculations. The following tables list these routines by these functional areas: • Comparison • Case conversion • Modification • Sub-string • Copying Where appropriate, the tables also provide columns indicating whether the routine is case-sensitive, uses the current locale, and/or supports multi-byte character sets. ***Null-terminated string comparison routines***
Compiler Directives for Strings (see page 2122)	The following compiler directives affect character and string types. ***Compiler directives for strings***
Copying a File (see page 2123)	The runtime library does not provide any routines for copying a file. However, if you are writing Windows-only applications, you can directly call the Windows API *CopyFile* function to copy a file. Like most of the runtime library file routines, *CopyFile* takes a filename as a parameter, not a file handle. When copying a file, be aware that the file attributes for the existing file are copied to the new file, but the security attributes are not. *CopyFile* is also useful when moving files across drives because neither the *RenameFile* function nor the Windows API *MoveFile* function can rename or... more (see page 2123)
Declaring and Initializing Strings (see page 2123)	When you declare a string:
Deleting a File (see page 2124)	Deleting a file erases the file from the disk and removes the entry from the disk's directory. There is no corresponding operation to restore a deleted file, so applications should generally allow users to confirm before deleting files. To delete a file, pass the name of the file to the *DeleteFile* function:

File Date-time Routines (see page 2125)	The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on operating system date-time values. *FileAge* returns the date-and-time stamp of a file, or -1 if the file does not exist. *FileSetDate* sets the date-and-time stamp for a specified file, and returns zero on success or an error code on failure. *FileGetDate* returns a date-and-time stamp for the specified file or –1 if the handle is invalid. As with most of the file manipulating routines, *FileAge* uses a string filename. *FileGetDate* and *FileSetDate*, however, use a *Handle* type as a parameter. To get the file handle either: • Use the *FileOpen... more (see page 2125)*
Finding a File (see page 2125)	There are three routines used for finding a file: *FindFirst*, *FindNext*, and *FindClose*. *FindFirst* searches for the first instance of a filename with a given set of attributes in a specified directory. *FindNext* returns the next entry matching the name and attributes specified in a previous call to *FindFirst*. *FindClose* releases memory allocated by *FindFirst*. You should always use *FindClose* to terminate a *FindFirst*/*FindNext* sequence. If you want to know if a file exists, a *FileExists* function returns *True* if the file exists, *False* otherwise. The three file find routines take a *TSearchRec... more (see page 2125)*
Manipulating Files (see page 2127)	Several common file operations are built into the runtime library. The routines for working with files operate at a high level. For most routines, you specify the name of the file and the routine makes the necessary calls to the operating system for you. In some cases, you use file handles instead. **Warning:** Although the Delphi language is not case sensitive, the Linux operating system is. Be attentive to case when working with files in cross-platform applications. The following topics describe how to use runtime library routines to perform file manipulation tasks: • Deleting a file (see page 2124) • Finding a file (see page 2125) • Renaming a (see page 2130)... more (see page 2127)
Mixing and Converting String Types (see page 2127)	Short, long, and wide strings can be mixed in assignments and expressions, and the compiler automatically generates code to perform the necessary string type conversions. However, when assigning a string value to a short string variable, be aware that the string value is truncated if it is longer than the declared maximum length of the short string variable. Long strings are already dynamically allocated. If you use one of the built-in pointer types, such as *PAnsiString*, *PString*, or *PWideString*, remember that you are introducing another level of indirection. Be sure this is what you intend.
Passing a Local Variable as a PChar (see page 2128)	Consider the case where you have a local string variable that you need to initialize by calling a function that takes a *PChar*. One approach is to create a local **array of char** and pass it to the function, then assign that variable to the string:
Creating Drawing Spaces (see page 2128)	The *TCanvas* class is defined in the Graphics unit, and encapsulates a Windows device context. This class handles all drawing for forms, visual containers (such as panels) and the printer object (see Printing (see page 2129)). Using the canvas object, you need not worry about allocating pens, brushes, palettes, and so on—all the allocation and deallocation are handled for you. *TCanvas* includes a large number of primitive graphics routines to draw lines, shapes, polygons, fonts, etc. onto any control that contains a canvas. For example, here is a button event handler that draws a line from the upper left corner to... more (see page 2128)
Printing (see page 2129)	The VCL *TPrinter* object encapsulates details of Windows printers. To get a list of installed and available printers, use the *Printers* property. Both printer objects use a *TCanvas* (which is identical to the form's *TCanvas*) which means that anything that can be drawn on a form can be printed as well. To print an image, call the *BeginDoc* method followed by whatever canvas graphics you want to print (including text through the *TextOut* method) and send the job to the printer by calling the *EndDoc* method. This example uses a button and a memo on a form. When... more (see page 2129)
Using TRegistryIniFile (see page 2130)	Many 32-bit Windows applications store their information in the system Registry instead of ini files because the Registry is hierarchical and doesn't suffer from the size limitations of ini files. If you are accustomed to using ini files and want to move your configuration information to the Registry instead, you can use the *TRegistryIniFile* class. You may also want to use *TRegistryIniFile* in cross-platform applications if you want to use the system Registry on Windows and an ini file on Linux. You can write most of your application so that it uses the *TCustomIniFile* type. You... more (see page 2130)

Renaming a File (see page 2130)	To change a file name, use the *RenameFile* function:
Returning a PChar Local Variable (see page 2131)	A common error when working with *PChars* is to store a local variable in a data structure, or return it as a value. When your routine ends, the *PChar* disappears because it is a pointer to memory, and not a reference counted copy of the string. For example:
String to PChar Conversions (see page 2131)	Long string (**AnsiString** and **UnicodeString**) to *PChar* conversions are not automatic. Some of the differences between strings and *PChars* can make conversions problematic: • Long strings are reference-counted, while *PChars* are not. • Assigning to a string copies the data, while a *PChar* is a pointer to memory. • Long strings are null-terminated and also contain the length of the string and other information, while *PChars* are simply null-terminated. Situations in which these differences can cause subtle errors are discussed in the following topics: • String dependencies (see page 2108) • Returning a PChar local variable (see page 2131) • Passing a local variable as a PChar (see page 2128)
Using TIniFile and TMemIniFile (see page 2131)	The ini file format is still popular, many configuration files (such as the DSK Desktop settings file) are in this format. This format is especially useful in cross-platform applications, where you can't always count on a system Registry for storing configuration information. The VCL/RTL provides two classes, *TIniFile* and *TMemIniFile*, to make reading and writing ini files very easy. *TIniFile* works directly with the ini file on disk while *TMemIniFile* buffers all changes in memory and does not write them to disk until you call the *UpdateFile* method. When you instantiate the *TIniFile* or *TMemIniFile* object, you pass... more (see page 2131)
Using File Streams (see page 2133)	The *TFileStream* class enables applications to read from and write to a file on disk. Because *TFileStream* is a stream object, it shares the common stream methods. You can use these methods to read from or write to the file, copy data to or from other stream classes, and read or write components values. See Using streams (see page 2099) for details on the capabilities that files streams inherit by being stream classes. In addition, file streams give you access to the file handle, so that you can use them with global file handling routines that require the file handle.
Accessing a Particular String (see page 2134)	The *Strings* array property contains the strings in the list, referenced by a zero-based index. Because *Strings* is the default property for string lists, you can omit the *Strings* identifier when accessing the list; thus
Adding a String to a List (see page 2134)	To add a string to the end of a string list, call the *Add* method, passing the new string as the parameter. To insert a string into the list, call the *Insert* method, passing two parameters: the string and the index of the position where you want it placed. For example, to make the string "Three" the third string in a list, you would use:
Associating Objects with a String List (see page 2135)	In addition to the strings stored in its *Strings* property, a string list can maintain references to *objects*, which it stores in its *Objects* property. Like *Strings*, *Objects* is an array with a zero-based index. The most common use for *Objects* is to associate bitmaps with strings for owner-draw controls. Use the *AddObject* or *InsertObject* method to add a string and an associated object to the list in a single step. *IndexOfObject* returns the index of the first string in the list associated with a specified object. Methods like *Delete*, *Clear*, and *Move* operate on both... more (see page 2135)
Copying a Complete String List (see page 2135)	You can use the *Assign* method to copy strings from a source list to a destination list, overwriting the contents of the destination list. To append strings without overwriting the destination list, use *AddStrings*. For example,
Counting the Strings in a List (see page 2135)	The read-only *Count* property returns the number of strings in the list. Since string lists use zero-based indexes, *Count* is one more than the index of the last string.

Creating a New String List (see page 2135)	A string list is typically part of a component. There are times, however, when it is convenient to create independent string lists, for example to store strings for a lookup table. The way you create and manage a string list depends on whether the list is short-term (constructed, used, and destroyed in a single routine) or long-term (available until the application shuts down). Whichever type of string list you create, remember that you are responsible for freeing the list when you finish with it.
Deleting a String from a List (see page 2137)	To delete a string from a string list, call the list's *Delete* method, passing the index of the string you want to delete. If you don't know the index of the string you want to delete, use the *IndexOf* method to locate it. To delete all the strings in a string list, use the *Clear* method. The following example uses *IndexOf* and *Delete* to find and delete a string:
Locating Items in a String List (see page 2138)	To locate a string in a string list, use the *IndexOf* method. *IndexOf* returns the index of the first string in the list that matches the parameter passed to it, and returns –1 if the parameter string is not found. *IndexOf* finds exact matches only; if you want to match partial strings, you must iterate through the string list yourself. For example, you could use *IndexOf* to determine whether a given file name is found among the Items of a list box:
Iterating Through Strings in a List (see page 2138)	To iterate through the strings in a list, use a **for** loop that runs from zero to *Count* –1. The following example converts each string in a list box to uppercase characters.
Loading and Saving String Lists (see page 2138)	String-list objects provide *SaveToFile* and *LoadFromFile* methods that let you store a string list in a text file and load a text file into a string list. Each line in the text file corresponds to a string in the list. Using these methods, you could, for example, create a simple text editor by loading a file into a memo component, or save lists of items for combo boxes. The following example loads a copy of the MyFile.ini file into a memo field and makes a backup copy called MyFile.bak.
Manipulating Strings in a List (see page 2139)	Operations commonly performed on string lists include: • Counting the strings in a list (see page 2135) • Accessing a particular string (see page 2134) • Finding the position of a string in the list (see page 2138) • Iterating through strings in a list (see page 2138) • Adding a string to a list (see page 2134) • Moving a string within a list • Deleting a string from a list (see page 2137) • Copying a complete string list (see page 2135)
Working with Lists (see page 2139)	The VCL/RTL includes many classes that represents lists or collections of items. They vary depending on the types of items they contain, what operations they support, and whether they are persistent. The following table lists various list classes, and indicates the types of items they contain:
Working with String Lists (see page 2140)	One of the most commonly used types of list is a list of character strings. Examples include items in a combo box, lines in a memo, names of fonts, and names of rows and columns in a string grid. The VCL/RTL provides a common interface to any list of strings through an object called *TStrings* and its descendants such as *TStringList* and *THashedStringList*. *TStringList* implements the abstract properties and methods introduced by *TStrings*, and introduces properties, events, and methods to • Sort the strings in the list. • Prohibit duplicate strings in sorted lists. • Respond to changes in the contents... more (see page 2140)

Wide Character Routines (see page 2141)	Wide strings contain two bytes per element and are used in a variety of situations. You may also choose to use wide strings because they simplify some of the string-handling issues in applications that have multiple target locales. Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems, which are single byte strings. For wide strings, there is a direct relationship between the number of bytes in the string and the number of elements in the string. In a MBCS string, you have... more (see page 2141)
Working with Files (see page 2142)	The VCL/RTL supports several ways of working with files. In addition to using file streams, there are several runtime library routines for performing file I/O. Both file streams and the global routines for reading from and writing to files are described in Approaches to file I/O (see page 2091). In addition to input/output operations, you may want to manipulate files on disk. Support for operations on the files themselves rather than their contents is described in Manipulating files (see page 2127). **Note:** When writing cross-platform applications, remember that although the Delphi language is not case sensitive, the Linux operating system is. When using objects... more (see page 2142)
Working with Strings (see page 2142)	The runtime library provides many specialized string-handling routines specific to a string type. These are routines for strings (`AnsiString` and `UnicodeString`), wide strings (`WideString`), and null-terminated strings (`PChars`). Routines that deal with null-terminated strings use the null-termination to determine the length of the string. There are no categories of routines listed for *ShortString* types. However, some built-in compiler routines deal with the *ShortString* type. These include, for example, the `Low` and `High` standard functions. For more details about the various string types, see the *Delphi Language Guide*. The following topics provide an overview... more (see page 2142)

3.2.4.10.1 Using the VCL/RTL: Overview

There are a number of units in the component library that provide the underlying support for most of the component libraries. These units include the global routines that make up the runtime library, a number of utility classes such as those that represent streams and lists, and the classes *TObject*, *TPersistent*, and *TComponent*. Collectively, these units are called the VCL/RTL. The VCL/RTL does not include any of the components that appear on the **Tool Palette**. Rather, the classes and routines in the VCL/RTL are used by the components that do appear on the **Tool Palette** and are available for you to use in application code or when you are writing your own classes.

The following topics discuss many of the classes and routines that make up the VCL/RTL and illustrate how to use them.

- Using streams (see page 2099)
- Working with files (see page 2142)
- Working with .ini files (see page 2099)
- Working with lists (see page 2139)
- Working with string lists (see page 2140)
- Working with strings (see page 2142)
- Creating drawing spaces (see page 2128)
- Printing (see page 2129)
- Converting measurements (see page 2092)
- Defining custom variants (see page 2096)

Note: This list of tasks is not exhaustive. The runtime library in the VCL/RTL contains many routines to perform tasks that are not mentioned here. These include a host of mathematical functions (defined in the Math unit), routines for working with date/time values (defined in the SysUtils and DateUtils units), and routines for working with Variant values (defined in the Variants unit).

See Also

Understanding the Component Library (see page 2055)

Overview of Component Creation (see page 1313)

3.2.4.10.2 Adding New Measurement Types

If you want to perform conversions between measurement units not already defined in the StdConvs unit, you need to create a new conversion family to represent the measurement units (*TConvType* values). When two *TConvType* values are registered with the same conversion family, the *Convert* function can convert between measurements made using the units represented by those *TConvType* values.

You first need to obtain *TConvFamily* values by registering a conversion family using the RegisterConversionFamily function. After you get a *TConvFamily* value (by registering a new conversion family or using one of the global variables in the StdConvs unit), you can use the RegisterConversionType function to add the new units to the conversion family. The following examples show how to do this:

Creating a simple conversion family and adding units (see page 2095)

Using a conversion function (see page 2113)

Using a class to manage conversions (see page 2110)

For more examples, refer to the source code for the standard conversions unit (stdconvs.pas). (Note that the source is not included in all editions of Delphi.)

See Also

Performing Conversions (see page 2105)

3.2.4.10.3 Approaches to File I/O

There are several approaches you can take when reading from and writing to files:

- The recommended approach for working with files is to use file streams. File streams are instances of the *TFileStream* class used to access information in disk files. File streams are a portable and high-level approach to file I/O. Because file streams make the file handle available, this approach can be combined with the next one. The Using file streams (see page 2133) discusses *TFileStream* in detail.
- You can work with files using a handle-based approach. File handles are provided by the operating system when you create or open a file to work with its contents. The SysUtils unit defines a number of file-handling routines that work with files using file handles. On Windows, these are typically wrappers around Windows API functions. Because the VCL/RTL functions can use the Delphi language syntax, and occasionally provide default parameter values, they are a convenient interface to the Windows API. Furthermore, there are corresponding versions on Linux, so you can use these routines in cross-platform applications. To use a handle-based approach, you first open a file using the *FileOpen* function or create a new file using the *FileCreate* function. Once you have the handle, use handle-based routines to work with its contents (write a line, read text, and so on).
- The System unit defines a number of file I/O routines that work with file variables, usually of the format "F: Text:" or "F: File:" File variables can have one of three types: typed, text, and untyped. A number of file-handling routines, such as *AssignPrn* and *writeln*, use them. The use of file variables is deprecated, and these file types are supported only for backward compatibility. They are incompatible with Windows file handles.

See Also

Manipulating Files (see page 2127)

3.2.4.10.4 Common List Operations

Although the various list classes contain different types of items and have different ancestries, most of them share a common set of methods for adding, deleting, rearranging, and accessing the items in the list.

Adding list items

Most list classes have an *Add* method, which lets you add an item to the end of the list (if it is not sorted) or to its appropriate position (if the list is sorted). Typically, the *Add* method takes as a parameter the item you are adding to the list and returns the position in the list where the item was added. In the case of bucket lists (*TBucketList* and *TObjectBucketList*), *Add* takes not only the item to add, but also a datum you can associate with that item. In the case of collections, *Add* takes no parameters, but creates a new item that it adds. The *Add* method on collections returns the item it added, so that you can assign values to the new item's properties.

Some list classes have an *Insert* method in addition to the *Add* method. Insert works the same way as the *Add* method, but has an additional parameter that lets you specify the position in the list where you want the new item to appear. If a class has an Add method, it also has an Insert method unless the position of items is predetermined For example, you can't use *Insert* with sorted lists because items must go in sort order, and you can't use *Insert* with bucket lists because the hash algorithm determines the item position.

The only classes that do not have an *Add* method are the ordered lists. Ordered lists are queues and stacks. To add items to an ordered list, use the *Push* method instead. *Push*, like *Add*, takes an item as a parameter and inserts it in the correct position.

Deleting list items

To delete a single item from one of the list classes, use either the *Delete* method or the *Remove* method. *Delete* takes a single parameter, the index of the item to remove. *Remove* also takes a single parameter, but that parameter is a reference to the item to remove, rather than its index. Some list classes support only a *Delete* method, some support only a *Remove* method, and some have both.

As with adding items, ordered lists behave differently than all other lists. Instead of using a *Delete* or *Remove* method, you remove an item from an ordered list by calling its *Pop* method. *Pop* takes no arguments, because there is only one item that can be removed.

If you want to delete all of the items in the list, you can call the *Clear* method. Clear is available for all lists except ordered lists.

Accessing list items

All list classes (except *TThreadList* and the ordered lists)*have a property that lets you access the items in the list. Typically, this property is called Items. For string lists, the property is called Strings, and for bucket lists it is called Data. The Items, Strings, or Data property is an indexed property, so that you can specify which item you want to access.*

On TThreadList, you must lock the list before you can access items. When you lock the list, the LockList method returns a TList object that you can use to access the items.

Ordered lists only let you access the "top" item of the list. You can obtain a reference to this item by calling the Peek method.

Rearranging list items

Some list classes have methods that let you rearrange the items in the list. Some have an *Exchange* method, that swaps the position of two items. Some have a *Move* method that lets you move an item to a specified location. Some have a *Sort* method that lets you sort the items in the list.

To see what methods are available, check the online Help for the list class you are using.

See Also

Working with Lists (see page 2139)

3.2.4.10.5 Converting Measurements

The ConvUtils unit declares a general-purpose Conversion Function (see page 2113) that you can use to convert a measurement from one set of units to another. You can perform conversions between compatible units of measurement such as feet and inches or days and weeks. Units that measure the same types of things are said to be in the same *conversion family*.

The units you're converting must be in the same conversion family. For information on doing conversions, see Performing Conversions (see page 2105).

The StdConvs unit defines several conversion families and measurement units within each family. In addition, you can create customized conversion families and associated units using the RegisterConversionTypeand RegisterConversionFamily functions. For information on extending conversion and conversion units, see Adding new measurement types (see page 2091).

3.2.4.10.6 Copying and Clearing Custom Variants

In addition to typecasting and the implementation of operators, you must indicate how to copy and clear variants of your custom Variant type.

To indicate how to copy the variant's value, implement the Copy method. Typically, this is an easy operation, although you must remember to allocate memory for any classes or structures you use to hold the variant's value:

```
procedure TComplexVariantType.Copy(var Dest: TVarData; const Source: TVarData;
const Indirect: Boolean);
begin
if Indirect and VarDataIsByRef(Source) then
VarDataCopyNoInd(Dest, Source)
else
with TComplexVarData(Dest) do
begin
VType := VarType;
VComplex := TComplexData.Create(TComplexVarData(Source).VComplex);
end;
end;
```

Note: The *Indirect* parameter in the *Copy* method signals that the copy must take into account the case when the variant holds only an indirect reference to its data.

Tip: If your custom variant type does not allocate any memory to hold its data (if the data fits entirely in the *TVarData* record), your implementation of the Copy method can simply call the SimplisticCopy method.

To indicate how to clear the variant's value, implement the Clear method. As with the Copy method, the only tricky thing about doing this is ensuring that you free any resources allocated to store the variant's data:

```
procedure TComplexVariantType.Clear(var V: TVarData);
begin
V.VType := varEmpty;
FreeAndNil(TComplexVarData(V).VComplex);
end;
```

You will also need to implement the IsClear method. This way, you can detect any invalid values or special values that represent "blank" data:

```
function TComplexVariantType.IsClear(const V: TVarData): Boolean;
begin
Result := (TComplexVarData(V).VComplex = nil) or
TComplexVarData(V).VComplex.IsZero;
end;
```

See Also

Implementing Binary Operations (see page 2100)

Implementing Comparison Operations (see page 2102)

Implementing Unary Operations (see page 2103)

Enabling Casting (see page 2097)

Using the TCustomVariantType Descendant (see page 2116)

Loading and Saving Custom Variant Values (see page 2104)

3.2.4.10.7 Copying Data from One Stream to Another

When copying data from one stream to another, you do not need to explicitly read and then write the data. Instead, you can use the *CopyFrom* method, as illustrated in the following example.

In the following example, one file is copied to another one using streams. The application includes two edit controls (*EdFrom* and *EdTo*) and a *Copy File* button.

```
procedure TForm1.CopyFileClick(Sender: TObject);
var
Source, Destination:TStream;
begin
Source := TFileStream.Create(edFrom.Text, fmOpenRead or fmShareDenyWrite);
try
Destination := TFileStream.Create(edTo.Text, fmOpenCreate or fmShareDenyRead);
try
Destination.CopyFrom(Source,Source.Size);
    finally
Destination.Free;
    end;
  finally
    Source.Free
end;
void __fastcall TForm1::CopyFileClick(TObject *Sender)
{
TStream* Source= new TFileStream(edFrom->Text, fmOpenRead | fmShareDenyWrite);
try
  {
    TStream* Destination = new TFileStream(edTo->Text, fmCreate | fmShareDenyRead);
try
    {
      Destination -> CopyFrom(Source, Source->Size);
    }
    __finally
    {
      delete Destination;
    }
  }
  __finally
  {
    delete Source;
  }
}
```

See Also

Using Streams to Read or Write Data (see page 2115)

Specifying the Stream Position and Size (see page 2106)

3.2.4.10.8 Creating a Class to Enable the Custom Variant Type

Custom variants work by using a special helper class that indicates how variants of the custom type can perform standard operations. You create this helper class by writing a descendant of TCustomVariantType. This involves overriding the appropriate virtual methods of *TCustomVariantType*.

The following topics provide details on how to implement and use a TCustomVariantType descendant:

- Enabling casting (see page 2097)

- Implementing binary operations (see page 2100)
- Implementing comparison operations (see page 2102)
- Implementing unary operations (see page 2103)
- Copying and clearing custom variants (see page 2093)
- Loading and saving custom variant values (see page 2104)
- Using the TCustomVariantType descendant (see page 2116)

See Also

Defining Custom Variants (see page 2096)

Writing Utilities to Work with a Custom Variant Type (see page 2117)

Storing a Custom Variant Type's Data (see page 2107)

Supporting Properties and Methods in Custom Variants (see page 2108)

3.2.4.10.9 Creating a Simple Conversion Family and Adding Units

One example of when you could create a new conversion family and add new measurement types might be when performing conversions between long periods of time (such as months to centuries) where a loss of precision can occur.

To explain this further, the *cbTime* family uses a day as its base unit. The base unit is the one that is used when performing all conversions within that family. Therefore, all conversions must be done in terms of days. An inaccuracy can occur when performing conversions using units of months or larger (months, years, decades, centuries, millennia) because there is not an exact conversion between days and months, days and years, and so on. Months have different lengths; years have correction factors for leap years, leap seconds, and so on.

If you are only using units of measurement greater than or equal to months, you can create a more accurate conversion family with years as its base unit. This example creates a new conversion family called *cbLongTime*.

Declare variables

First, you need to declare variables for the identifiers. The identifiers are used in the new LongTime conversion family, and the units of measurement that are its members:

```
var
cbLongTime: TConvFamily;
ltMonths: TConvType;
ltYears: TConvType;
ltDecades: TConvType;
ltCenturies: TConvType;
ltMillennia: TConvType;
tConvFamily cbLongTime;
TConvType ltMonths;
TConvType ltYears;
TConvType ltDecades;
TConvType ltCenturies;
TConvType ltMillennia;
```

Register the conversion family

Next, register the conversion family:

```
cbLongTime := RegisterConversionFamily ('Long Times');
cbLongTime = RegisterConversionFamily ("Long Times");
```

Although an *UnregisterConversionFamily* procedure is provided, you don't need to unregister conversion families unless the unit that defines them is removed at runtime. They are automatically cleaned up when your application shuts down.

Register measurement units

Next, you need to register the measurement units within the conversion family that you just created. You use the *RegisterConversionType* function, which registers units of measurement within a specified family. You need to define the base unit which in the example is years, and the other units are defined using a factor that indicates their relation to the base unit. So, the factor for *ltMonths* is 1/12 because the base unit for the LongTime family is years. You also include a description of the units to which you are converting.

The code to register the measurement units is shown here:

```
ltMonths:=RegisterConversionType(cbLongTime,'Months',1/12);
ltYears:=RegisterConversionType(cbLongTime,'Years',1);
ltDecades:=RegisterConversionType(cbLongTime,'Decades',10);
ltCenturies:=RegisterConversionType(cbLongTime,'Centuries',100);
ltMillennia:=RegisterConversionType(cbLongTime,'Millennia',1000);
ltMonths = RegisterConversionType(cbLongTime,"Months",1/12);
ltYears = RegisterConversionType(cbLongTime,"Years",1);
ltDecades = RegisterConversionType(cbLongTime,"Decades",10);
ltCenturies = RegisterConversionType(cbLongTime,"Centuries",100);
ltMillennia = RegisterConversionType(cbLongTime,"Millennia",1000);
```

Use the new units

You can now use the newly registered units to perform conversions. The global *Convert* function can convert between any of the conversion types that you registered with the *cbLongTime* conversion family.

So instead of using the following *Convert* call,

```
Convert(StrToFloat(Edit1.Text),tuMonths,tuMillennia);
Convert(StrToFloat(Edit1->Text),tuMonths,tuMillennia);
```

you can now use this one for greater accuracy:

```
Convert(StrToFloat(Edit1.Text),ltMonths,ltMillennia);
Convert(StrToFloat(Edit1->Text),ltMonths,ltMillennia);
```

See Also

Performing Conversions (see page 2105)

3.2.4.10.10 Defining Custom Variants

One powerful built-in type of the Delphi language is the Variant type. Variants represent values whose type is not determined at compile time. Instead, the type of their value can change at runtime. Variants can mix with other variants and with integer, real, string, and boolean values in expressions and assignments; the compiler automatically performs type conversions.

By default, variants can't hold values that are records, sets, static arrays, files, classes, class references, or pointers. You can, however, extend the Variant type to work with any particular example of these types. All you need to do is create a descendant of the *TCustomVariantType* class that indicates how the Variant type performs standard operations.

To create a Variant type:

1. Map the storage of the variant's data (see page 2107) on to the *TVarData* record.
2. Declare a class that descends from TCustomVariantType (see page 2094). Implement all required behavior (including type conversion rules) in the new class.
3. Write utility methods (see page 2117) for creating instances of your custom variant and recognizing its type.

The above steps extend the Variant type so that the standard operators work with your new type and the new Variant type can be cast to other data types. You can further enhance your new Variant type so that it supports properties and methods that you define. When creating a Variant type that supports properties or methods (see page 2108), you use *TInvokeableVariantType* or *TPublishableVariantType* as a base class rather than *TCustomVariantType*.

3.2.4.10.11 Enabling Casting

One of the most important features of the custom variant type for you to implement is typecasting. The flexibility of variants arises, in part, from their implicit typecasts.

There are two methods for you to implement that enable the custom Variant type to perform typecasts: Cast, which converts another variant type to your custom variant, and CastTo, which converts your custom Variant type to another type of Variant.

When implementing either of these methods, it is relatively easy to perform the logical conversions from the built-in variant types. You must consider, however, the possibility that the variant to or from which you are casting may be another custom Variant type. To handle this situation, you can try casting to one of the built-in Variant types as an intermediate step.

For example, the following *Cast* method, from the *TComplexVariantType* class uses the type Double as an intermediate type:

```
procedure TComplexVariantType.Cast(var Dest: TVarData; const Source: TVarData);
var
LSource, LTemp: TVarData;
begin
VarDataInit(LSource);
try
VarDataCopyNoInd(LSource, Source);
if VarDataIsStr(LSource) then
TComplexVarData(Dest).VComplex := TComplexData.Create(VarDataToStr(LSource))
else
begin
VarDataInit(LTemp);
try
VarDataCastTo(LTemp, LSource, varDouble);
TComplexVarData(Dest).VComplex := TComplexData.Create(LTemp.VDouble, 0);
finally
VarDataClear(LTemp);
end;
end;
Dest.VType := VarType;
finally
VarDataClear(LSource);
end;
end;
```

In addition to the use of Double as an intermediate Variant type, there are a few things to note in this implementation:

- The last step of this method sets the *VType* member of the returned *TVarData* record. This member gives the Variant type code. It is set to the VarType property of *TComplexVariantType*, which is the Variant type code assigned to the custom variant.
- The custom variant's data (*Dest*) is typecast from *TVarData* to the record type that is actually used to store its data (see page 2107) (*TComplexVarData*). This makes the data easier to work with.
- The method makes a local copy of the source variant rather than working directly with its data. This prevents side effects that may affect the source data.

When casting from a complex variant to another type, the *CastTo* method also uses an intermediate type of Double (for any destination type other than a string):

```
procedure TComplexVariantType.CastTo(var Dest: TVarData; const Source: TVarData;
const AVarType: TVarType);
var
LTemp: TVarData;
begin
if Source.VType = VarType then
case AVarType of
varOleStr:
VarDataFromOleStr(Dest, TComplexVarData(Source).VComplex.AsString);
varString:
VarDataFromStr(Dest, TComplexVarData(Source).VComplex.AsString);
else
VarDataInit(LTemp);
try
LTemp.VType := varDouble;
LTemp.VDouble := TComplexVarData(LTemp).VComplex.Real;
VarDataCastTo(Dest, LTemp, AVarType);
finally
VarDataClear(LTemp);
end;
end
else
RaiseCastError;
end;
```

Note that the *CastTo* method includes a case where the source variant data does not have a type code that matches the *VarType* property. This case only occurs for empty (unassigned) source variants.

See Also

Implementing Binary Operations (see page 2100)

Implementing Comparison Operations (see page 2102)

Implementing Unary Operations (see page 2103)

Copying and Clearing Custom Variants (see page 2093)

Using the TCustomVariantType Descendant (see page 2116)

Loading and Saving Custom Variant Values (see page 2104)

3.2.4.10.12 Use __fastcall When Overriding VCL Methods

Member functions for VCL classes generally use __fastcall in their declaration unless declared otherwise. This means that you must use __fastcall in the declaration of any VCL class member function that you override.

See Also

__fastcall (see page 530)

__msfastcall (see page 526)

3.2.4.10.13 Working with ini Files and the System Registry

Many applications use ini files to store configuration information. The VCL/RTL includes two classes for working with ini files: *TIniFile* and *TMemIniFile*. Using ini files has the advantage that they can be used in cross-platform applications and they are easy to read and edit. For information on these classes, see Using TIniFile and TMemIniFile (see page 2131) for more information.

Many Windows applications replace the use of ini files with the system Registry. The Windows system Registry is a hierarchical database that acts as a centralized storage space for configuration information. The VCL includes classes for working with the system Registry. Two of these classes, *TRegistryIniFile* and *TRegistry,* are discussed here because of their similarity to the classes for working with ini files.

TRegistryIniFile is useful for cross-platform applications, because it shares a common ancestor (*TCustomIniFile*) with the classes that work with ini files. If you confine yourself to the methods of the common ancestor (*TCustomIniFile*) your application can work on both applications with a minimum of conditional code. *TRegistryIniFile* is discussed in Using TRegistryIniFile (see page 2130).

For applications that are not cross-platform, you can use the *TRegistry* class. The properties and methods of *TRegistry* have names that correspond more directly to the way the system Registry is organized, because it does not need to be compatible with the classes for ini files. *TRegistry* is discussed in Using TRegistry (see page 2099).

See Also

Using Streams (see page 2099)

Working with Files (see page 2142)

Working with Strings (see page 2142)

Working with String Lists (see page 2140)

3.2.4.10.14 Using Streams

Streams are classes that let you read and write data. They provide a common interface for reading and writing to different media such as memory, strings, sockets, and BLOB fields in databases. There are several stream classes, which all descend from *TStream*. Each stream class is specific to one media type. For example, *TMemoryStream* reads from or writes to a memory image; *TFileStream* reads from or writes to a file.

The following topics describe the methods common to all stream classes:

- Using streams to read or write data (see page 2115)
- Copying data from one stream to another (see page 2094)
- Specifying the stream position and size (see page 2106)

See Also

TStream

Working with Files (see page 2142)

Using File Streams (see page 2133)

3.2.4.10.15 Using TRegistry

If you are writing a Windows-only application and are comfortable with the structure of the system Registry, you can use

TRegistry. Unlike *TRegistryIniFile*, which uses the same properties and methods of other ini file components, the properties and methods of *TRegistry* correspond more directly to the structure of the system Registry. You can specify both the root key and subkey using *TRegistry*, while *TRegistryIniFile* uses `HKEY_CURRENT_USER` as the root key.

In addition to methods for opening, closing, saving, moving, copying, and deleting keys, *TRegistry* lets you specify the access level you want to use.

Note: *TRegistry* is not available for cross-platform programming.

The following example retrieves a value from a registry entry:

```
function GetRegistryValue(KeyName: string): string;
var
  Registry: TRegistry;
begin
Registry := TRegistry.Create(KEY_READ);
  try
    Registry.RootKey = HKEY_LOCAL_MACHINE;
// False because we do not want to create it if it doesn't exist
    Registry.OpenKey(KeyName, False);
    Result := Registry.ReadString('VALUE1');
  finally
    Registry.Free;
  end;
end;
#include <Registry.hpp>
AnsiString GetRegistryValue(AnsiString KeyName)
{
AnsiString S;
TRegistry *Registry = new TRegistry(KEY_READ);
try
{
Registry->RootKey = HKEY_LOCAL_MACHINE;
// False because we do not want to create it if it doesn't exist
Registry->OpenKey(KeyName,false);
S = Registry->ReadString("VALUE1");
}
__finally
{
delete Registry;
}
return S;
}
```

See Also

Using TRegistryIniFile (see page 2130)

3.2.4.10.16 Implementing Binary Operations

To allow the custom variant type to work with standard binary operators (+, -, *, /, div, mod, shl, shr, and, or, xor listed in the System unit), you must override the BinaryOp method. *BinaryOp* has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the operator. Implement this method to perform the operation and return the result using the same variable that contained the left-hand operand.

For example, the following *BinaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:

```
procedure TComplexVariantType.BinaryOp(var Left: TVarData; const Right: TVarData;
  const Operator: TVarOp);
begin
  if Right.VType = VarType then
    case Left.VType of
      varString:
        case Operator of
```

```
          opAdd: Variant(Left) := Variant(Left) + TComplexVarData(Right).VComplex.AsString;
        else
          RaiseInvalidOp;
        end;
      else
        if Left.VType = VarType then
          case Operator of
            opAdd:
              TComplexVarData(Left).VComplex.DoAdd(TComplexVarData(Right).VComplex);
            opSubtract:
              TComplexVarData(Left).VComplex.DoSubtract(TComplexVarData(Right).VComplex);
            opMultiply:
              TComplexVarData(Left).VComplex.DoMultiply(TComplexVarData(Right).VComplex);
            opDivide:
              TComplexVarData(Left).VComplex.DoDivide(TComplexVarData(Right).VComplex);
            else
              RaiseInvalidOp;
          end
        else
          RaiseInvalidOp;
    end
  else
    RaiseInvalidOp;
end;
```

There are several things to note in this implementation:

This method only handles the case where the variant on the right side of the operator is a custom variant that represents a complex number. If the left-hand operand is a complex variant and the right-hand operand is not, the complex variant forces the right-hand operand first to be cast to a complex variant. It does this by overriding the RightPromotion method so that it always requires the type in the *VarType* property:

```
function TComplexVariantType.RightPromotion(const V: TVarData;
const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
{ Complex Op TypeX }
RequiredVarType := VarType;
Result := True;
end;
```

The addition operator is implemented for a string and a complex number (by casting the complex value to a string and concatenating), and the addition, subtraction, multiplication, and division operators are implemented for two complex numbers using the methods of the *TComplexData* object that is stored in the complex variant's data. This is accessed by casting the *TVarData* record to a *TComplexVarData* record and using its *VComplex* member.

Attempting any other operator or combination of types causes the method to call the *RaiseInvalidOp* method, which causes a runtime error. The *TCustomVariantType* class includes a number of utility methods such as *RaiseInvalidOp* that can be used in the implementation of custom variant types.

BinaryOp only deals with a limited number of types: strings and other complex variants. It is possible, however, to perform operations between complex numbers and other numeric types. For the *BinaryOp* method to work, the operands must be cast to complex variants before the values are passed to this method. We have already seen (above) how to use the *RightPromotion* method to force the right-hand operand to be a complex variant if the left-hand operand is complex. A similar method, LeftPromotion, forces a cast of the left-hand operand when the right-hand operand is complex:

```
function TComplexVariantType.LeftPromotion(const V: TVarData;
const Operator: TVarOp; out RequiredVarType: TVarType): Boolean;
begin
{ TypeX Op Complex }
if (Operator = opAdd) and VarDataIsStr(V) then
RequiredVarType := varString
else
RequiredVarType := VarType;
  Result := True;
```

```
end;
```

This *LeftPromotion* method forces the left-hand operand to be cast to another complex variant, unless it is a string and the operation is addition, in which case *LeftPromotion* allows the operand to remain a string.

See Also

Implementing Comparison Operations (see page 2102)

Implementing Unary Operations (see page 2103)

Enabling Casting (see page 2097)

Copying and Clearing Custom Variants (see page 2093)

Using the TCustomVariantType Descendant (see page 2116)

Loading and Saving Custom Variant Values (see page 2104)

3.2.4.10.17 Implementing Comparison Operations

There are two ways to enable a custom variant type to support comparison operators (=, <>, <, <=, >, >=). You can override the *Compare* method, or you can override the *CompareOp* method.

The Compare method is easiest if your custom variant type supports the full range of comparison operators. *Compare* takes three parameters: the left-hand operand, the right-hand operand, and a var Parameter that returns the relationship between the two. For example, the *TConvertVariantType* object in the VarConv unit implements the following *Compare* method:

```
procedure TConvertVariantType.Compare(const Left, Right: TVarData;
var Relationship: TVarCompareResult);
const
CRelationshipToRelationship: array [TValueRelationship] of TVarCompareResult =
(crLessThan, crEqual, crGreaterThan);
var
LValue: Double;
LType: TConvType;
LRelationship: TValueRelationship;
begin
// supports...
//   convvar cmp number
//     Compare the value of convvar and the given number
  //   convvar1 cmp convvar2
//     Compare after converting convvar2 to convvar1's unit type
  //  The right can also be a string.  If the string has unit info then it is
//    treated like a varConvert else it is treated as a double
LRelationship := EqualsValue;
case Right.VType of
    varString:
      if TryStrToConvUnit(Variant(Right), LValue, LType) then
        if LType = CIllegalConvType then
          LRelationship := CompareValue(TConvertVarData(Left).VValue, LValue)
        else
          LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                                                TConvertVarData(Left).VConvType, LValue, LType)
        else
          RaiseCastError;
    varDouble:
      LRelationship := CompareValue(TConvertVarData(Left).VValue, TVarData(Right).VDouble);
    else
      if Left.VType = VarType then
        LRelationship := ConvUnitCompareValue(TConvertVarData(Left).VValue,
                         TConvertVarData(Left).VConvType, TConvertVarData(Right).VValue,
                         TConvertVarData(Right).VConvType)
      else
        RaiseInvalidOp;
```

```
  end;
  Relationship := CRelationshipToRelationship[LRelationship];
end;
```

If the custom type does not support the concept of "greater than" or "less than," only "equal" or "not equal," however, it is difficult to implement the *Compare* method, because *Compare* must return *crLessThan*, *crEqual*, or *crGreaterThan*. When the only valid response is "not equal," it is impossible to know whether to return *crLessThan* or *crGreaterThan*. Thus, for types that do not support the concept of ordering, you can override the *CompareOp* method instead.

CompareOp has three parameters: the value of the left-hand operand, the value of the right-hand operand, and the comparison operator. Implement this method to perform the operation and return a boolean that indicates whether the comparison is *True*. You can then call the *RaiseInvalidOp* method when the comparison makes no sense.

For example, the following *CompareOp* method comes from the *TComplexVariantType* object in the VarCmplx unit. It supports only a test of equality or inequality:

```
function TComplexVariantType.CompareOp(const Left, Right: TVarData;
const Operator: Integer): Boolean;
begin
Result := False;
if (Left.VType = VarType) and (Right.VType = VarType) then
case Operator of
opCmpEQ:
Result := TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
opCmpNE:
Result := not TComplexVarData(Left).VComplex.Equal(TComplexVarData(Right).VComplex);
else
RaiseInvalidOp;
end
else
RaiseInvalidOp;
end;
```

Note that the types of operands that both these implementations support are very limited. As with binary operations (see page 2100), you can use the *RightPromotion* and *LeftPromotion* methods to limit the cases you must consider by forcing a cast before *Compare* or *CompareOp* is called.

See Also

Implementing Binary Operations (see page 2100)

Implementing Unary Operations (see page 2103)

Enabling Casting (see page 2097)

Copying and Clearing Custom Variants (see page 2093)

Using the TCustomVariantType Descendant (see page 2116)

Loading and Saving Custom Variant Values (see page 2104)

3.2.4.10.18 Implementing Unary Operations

To allow the custom variant type to work with standard unary operators (-, not), you must override the UnaryOp method. *UnaryOp* has two parameters: the value of the operand and the operator. Implement this method to perform the operation and return the result using the same variable that contained the operand.

For example, the following *UnaryOp* method comes from the *TComplexVariantType* defined in the VarCmplx unit:

```
procedure TComplexVariantType.UnaryOp(var Right: TVarData; const Operator: TVarOp);
begin
if Right.VType = VarType then
case Operator of
```

```
opNegate:
TComplexVarData(Right).VComplex.DoNegate;
else
RaiseInvalidOp;
end
else
RaiseInvalidOp;
end;
```

Note that for the logical **not** operator, which does not make sense for complex values, this method calls *RaiseInvalidOp* to cause a runtime error.

See Also

Implementing Binary Operations (see page 2100)

Implementing Comparison Operations (see page 2102)

Enabling Casting (see page 2097)

Copying and Clearing Custom Variants (see page 2093)

Using the TCustomVariantType Descendant (see page 2116)

Loading and Saving Custom Variant Values (see page 2104)

3.2.4.10.19 Loading and Saving Custom Variant Values

By default, when the custom variant is assigned as the value of a published property, it is typecast to a string when that property is saved to a form file, and converted back from a string when the property is read from a form file. You can, however, provide your own mechanism for loading and saving custom variant values in a more natural representation. To do so, the *TCustomVariantType* descendant must implement the IVarStreamable interface from Classes.pas.

IVarStreamable defines two methods, *StreamIn* and *StreamOut*, for reading a variant's value from a stream and for writing the variant's value to the stream. For example, *TComplexVariantType*, in the VarCmplx unit, implements the *IVarStreamable* methods as follows:

```
procedure TComplexVariantType.StreamIn(var Dest: TVarData; const Stream: TStream);
begin
with TReader.Create(Stream, 1024) do
try
with TComplexVarData(Dest) do
begin
VComplex := TComplexData.Create;
VComplex.Real := ReadFloat;
VComplex.Imaginary := ReadFloat;
end;
finally
Free;
end;
end;
procedure TComplexVariantType.StreamOut(const Source: TVarData; const Stream: TStream);
begin
with TWriter.Create(Stream, 1024) do
try
with TComplexVarData(Source).VComplex do
begin
WriteFloat(Real);
WriteFloat(Imaginary);
end;
finally
Free;
end;
end;
```

Note how these methods create a Reader or Writer object for the *Stream* parameter to handle the details of reading or writing values.

See Also

Implementing Binary Operations (see page 2100)

Implementing Comparison Operations (see page 2102)

Implementing Unary Operations (see page 2103)

Enabling Casting (see page 2097)

Copying and Clearing Custom Variants (see page 2093)

Using the TCustomVariantType Descendant (see page 2116)

3.2.4.10.20 Performing Conversions

You can use the *Convert* function to perform both simple and complex conversions. It includes a simple syntax and a second syntax for performing conversions between complex measurement types.

Performing simple conversions

You can use the *Convert* function to convert a measurement from one set of units to another. The *Convert* function converts between units that measure the same type of thing (distance, area, time, temperature, and so on).

To use *Convert*, you must specify the units from which to convert and to which to convert. You use the TConvType type to identify the units of measurement.

For example, this converts a temperature from degrees Fahrenheit to degrees Kelvin:

```
TempInKelvin := Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
TempInKelvin = Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

Performing complex conversions

You can also use the *Convert* function to perform more complex conversions between the ratio of two measurement types. Examples of when you might need to use this this are when converting miles per hour to meters per minute for calculating speed or when converting gallons per minute to liters per hour for calculating flow.

For example, the following call converts miles per gallon to kilometers per liter:

```
nKPL := Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
double nKPL = Convert(StrToFloat(Edit1.Text), duMiles, vuGallons, duKilometers, vuLiter);
```

The units you're converting must be in the same conversion family (they must measure the same thing). If the units are not compatible, *Convert* raises an *EConversionError* exception. You can check whether two *TConvType* values are in the same conversion family by calling CompatibleConversionTypes.

The StdConvs unit defines several families of *TConvType* values.

3.2.4.10.21 Persistent Lists

Persistent lists can be saved to a form file. Because of this, they are often used as the type of a published property on a component. You can add items to the list at design time, and those items are saved with the object so that they are there when the component that uses them is loaded into memory at runtime. There are two main types of persistent lists: string lists and collections.

Examples of string lists include *TStringList* and *THashedStringList*. String lists, as the name implies, contain strings. They provide special support for strings of the form Name=Value, so that you can look up the value associated with a name. In

addition, most string lists let you associate an object with each string in the list. String lists are described in more detail in Working with string lists (see page 2140).

Collections descend from the class *TCollection*. Each *TCollection* descendant is specialized to manage a specific class of items, where that class descends from *TCollectionItem*. Collections support many of the common list operations (see page 2091). All collections are designed to be the type of a published property, and many can not function independently of the object that uses them to implement on of its properties. At design time, the property whose value is a collection can use the collection editor to let you add, remove, and rearrange items. The collection editor provides a common user interface for manipulating collections.

See Also

Working with Lists (see page 2139)

3.2.4.10.22 Specifying the Stream Position and Size

In addition to methods for reading and writing, streams permit applications to seek to an arbitrary position in the stream or change the size of the stream. Once you seek to a specified position, the next read or write operation starts reading from or writing to the stream at that position.

Seeking to a specific position

The *Seek* method is the most general mechanism for moving to a particular position in the stream. There are two overloads for the *Seek* method:

```
function Seek(Offset: Longint; Origin: Word): Longint;
function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
virtual int __fastcall Seek(int Offset, Word Origin);
virtual __int64 __fastcall Seek(const __int64 Offset, TSeekOrigin Origin);
```

Both overloads work the same way. The difference is that one version uses a 32-bit integer to represent positions and offsets, while the other uses a 64-bit integer.

The *Origin* parameter indicates how to interpret the *Offset* parameter. *Origin* should be one of the following values:

Values for the Origin parameter

Value	Meaning
soFromBeginning	Offset is from the beginning of the resource. Seek moves to the position Offset. Offset must be >= 0.
soFromCurrent	Offset is from the current position in the resource. Seek moves to Position + Offset.
soFromEnd	Offset is from the end of the resource. Offset must be <= 0 to indicate a number of bytes before the end of the file.

Seek resets the current stream position, moving it by the indicated offset. *Seek* returns the new current position in the stream.

Using Position and Size properties

All streams have properties that hold the current position and size of the stream. These are used by the *Seek* method, as well as all the methods that read from or write to the stream.

The *Position* property indicates the current offset, in bytes, into the stream (from the beginning of the streamed data). The declaration for *Position* is:

```
property Position: Int64;
__property __int64 Position = {read=GetPosition, write=SetPosition, nodefault};
```

The *Size* property indicates the size of the stream in bytes. It can be used to determine the number of bytes available for reading, or to truncate the data in the stream. The declaration for *Size* is:

```
property Size: Int64;
__property __int64 Size = {read=GetSize, write=SetSize64, nodefault};
```

Size is used internally by routines that read and write to and from the stream.

Setting the *Size* property changes the size of the data in the stream. For example, on a file stream, setting *Size* inserts an end of file marker to truncate the file. If the *Size* of the stream cannot be changed, an exception is raised. For example, trying to change the *Size* of a read-only file stream raises an exception.

See Also

Using Streams to Read or Write Data (see page 2115)

Copying Data from One Stream to Another (see page 2094)

3.2.4.10.23 Storing a Custom Variant Type's Data

Variants store their data in the *TVarData* record type. This type is a record that contains 16 bytes. The first word indicates the type of the variant, and the remaining 14 bytes are available to store the data. While your new Variant type can work directly with a *TVarData* record, it is usually easier to define a record type whose members have names that are meaningful for your new type, and cast that new type onto the *TVarData* record type.

For example, the VarConv unit defines a custom variant type that represents a measurement. The data for this type includes the units (*TConvType*) of measurement, as well as the value (a double). The VarConv unit defines its own type to represent such a value:

```
TConvertVarData = packed record
  VType: TVarType;
  VConvType: TConvType;
  Reserved1, Reserved2: Word;
  VValue: Double;
end;
```

This type is exactly the same size as the *TVarData* record. When working with a custom variant of the new type, the variant (or its *TVarData* record) can be cast to *TConvertVarData*, and the custom Variant type simply works with the *TVarData* record as if it were a *TConvertVarData* type.

Note: When defining a record that maps onto the *TVarData* record in this way, be sure to define it as a packed record.

If your new custom Variant type needs more than 14 bytes to store its data, you can define a new record type that includes a pointer or object instance. For example, the VarCmplx unit uses an instance of the class *TComplexData* to represent the data in a complex-valued variant. It therefore defines a record type the same size as *TVarData* that includes a reference to a *TComplexData* object:

```
TComplexVarData = packed record
  VType: TVarType;
  Reserved1, Reserved2, Reserved3: Word;
  VComplex: TComplexData;
  Reserved4: LongInt;
end;
```

Object references are actually pointers (two Words), so this type is the same size as the *TVarData* record. As before, a complex custom variant (or its *TVarData* record), can be cast to *TComplexVarData*, and the custom variant type works with the *TVarData* record as if it were a *TComplexVarData* type.

See Also

Writing Utilities to Work with a Custom Variant Type (see page 2117)

Creating a Class to Enable the Custom Variant Type (see page 2094)

Supporting Properties and Methods in Custom Variants (see page 2108)

Defining Custom Variants (see page 2096)

3.2.4.10.24 String Dependencies

Sometimes you need to convert a long string to a null-terminated string, for example, if you are using a function that takes a *PChar*. If you must cast a string to a *PChar*, be aware that you are responsible for the lifetime of the resulting *PChar*. Because long strings are reference counted, typecasting a string to a *PChar* increases the dependency on the string by one, without actually incrementing the reference count. When the reference count hits zero, the string will be destroyed, even though there is an extra dependency on it. The cast *PChar* will also disappear, while the routine you passed it to may still be using it. For example:

```
procedure my_func(x: string);
begin
  // do something with x
  some_proc(PChar(x)); // cast the string to a PChar
  // you now need to guarantee that the string remains
  // as long as the some_proc procedure needs to use it
end;
```

See Also

Returning a PChar Local Variable (see page 2131)

Passing a Local Variable as a PChar (see page 2128)

3.2.4.10.25 Supporting Properties and Methods in Custom Variants

Some variants have properties and methods. For example, when the value of a variant is an interface, you can use the variant to read or write the values of properties on that interface and call its methods. Even if your custom variant type does not represent an interface, you may want to give it properties and methods that an application can use in the same way.

Using TInvokeableVariantType

To provide support for properties and methods, the class you create to enable the new custom variant type (see page 2094) should descend from TInvokeableVariantType instead of directly from *TCustomVariantType*.

TInvokeableVariantType defines four methods:

- DoFunction
- DoProcedure
- GetProperty
- SetProperty

that you can implement to support properties and methods on your custom variant type.

For example, the VarConv unit uses *TInvokeableVariantType* as the base class for *TConvertVariantType* so that the resulting custom variants can support properties. The following example shows the property getter for these properties:

```
function TConvertVariantType.GetProperty(var Dest: TVarData;
const V: TVarData; const Name: String): Boolean;
var
LType: TConvType;
begin
// supports...
//   'Value'
//   'Type'
//   'TypeName'
//   'Family'
//   'FamilyName'
//   'As[Type]'
Result := True;
if Name = 'VALUE' then
```

```
Variant(Dest) := TConvertVarData(V).VValue
else if Name = 'TYPE' then
Variant(Dest) := TConvertVarData(V).VConvType
else if Name = 'TYPENAME' then
Variant(Dest) := ConvTypeToDescription(TConvertVarData(V).VConvType)
else if Name = 'FAMILY' then
Variant(Dest) := ConvTypeToFamily(TConvertVarData(V).VConvType)
else if Name = 'FAMILYNAME' then
Variant(Dest) := ConvFamilyToDescription(ConvTypeToFamily(TConvertVarData(V).VConvType))
else if System.Copy(Name, 1, 2) = 'AS' then
begin
if DescriptionToConvType(ConvTypeToFamily(TConvertVarData(V).VConvType),
System.Copy(Name, 3, MaxInt), LType) then
VarConvertCreateInto(Variant(Dest), Convert(TConvertVarData(V).VValue,
TConvertVarData(V).VConvType, LType), LType)
else
Result := False;
end
else
Result := False;
end;
```

The *GetProperty* method checks the *Name* parameter to determine what property is wanted. It then retrieves the information from the *TVarData* record of the Variant (*V*), and returns it as a Variant (*Dest*). Note that this method supports properties whose names are dynamically generated at runtime (As[Type]), based on the current value of the custom variant.

Similarly, the *SetProperty*, *DoFunction*, and *DoProcedure* methods are sufficiently generic that you can dynamically generate method names, or respond to variable numbers and types of parameters.

Using TPublishableVariantType

If the custom variant type stores its data using an object instance, then there is an easier way to implement properties, as long as they are also properties of the object that represents the variant's data. If you use TPublishableVariantType as the base class for your custom variant type, then you need only implement the GetInstance method, and all the published properties of the object that represents the variant's data are automatically implemented for the custom variants.

For example, as was seen in Storing a custom variant type's data (see page 2107), *TComplexVariantType* stores the data of a complex-valued variant using an instance of *TComplexData*. *TComplexData* has a number of published properties (*Real*, *Imaginary*, *Radius*, *Theta*, and *FixedTheta*), that provide information about the complex value. *TComplexVariantType* descends from *TPublishableVariantType*, and implements the *GetInstance* method to return the *TComplexData* object (in TypInfo.pas) that is stored in a complex-valued variant's *TVarData* record:

```
function TComplexVariantType.GetInstance(const V: TVarData): TObject;
begin
Result := TComplexVarData(V).VComplex;
end;
```

TPublishableVariantType does the rest. It overrides the *GetProperty* and *SetProperty* methods to use the runtime type information (RTTI) of the *TComplexData* object for getting and setting property values.

Note: For *TPublishableVariantType* to work, the object that holds the custom variant's data must be compiled with RTTI. This means it must be compiled using the {$M+} compiler directive, or descend from *TPersistent*.

See Also

Storing a Custom Variant Type's Data (see page 2107)

Creating a Class to Enable the Custom Variant Type (see page 2094)

Writing Utilities to Work with a Custom Variant Type (see page 2117)

3.2.4.10.26 Using a Class to Manage Conversions

You can always use conversion functions to register a conversion unit. There are times, however, when this requires you to create an unnecessarily large number of functions that all do essentially the same thing.

If you can write a set of conversion functions that differ only in the value of a parameter or variable, you can create a class to handle those conversions. For example, there is a set standard techniques for converting between the various European currencies since the introduction of the Euro. Even though the conversion factors remain constant (unlike the conversion factor between, say, dollars and Euros), you can't use a simple conversion factor approach to properly convert between European currencies for two reasons:

- The conversion must round to a currency-specific number of digits.
- The conversion factor approach uses an inverse factor to the one specified by the standard Euro conversions.

However, this can all be handled by the conversion functions such as the following:

```
function FromEuro(const AValue: Double, Factor; FRound: TRoundToRange): Double;
begin
Result := RoundTo(AValue * Factor, FRound);
end;
function ToEuro(const AValue: Double, Factor): Double;
begin
Result := AValue / Factor;
end;
double __fastcall FromEuro(const double AValue, const double Factor, TRoundToRange FRound)
{
return(RoundTo(AValue * Factor, FRound));
}
double __fastcall ToEuro(const double AValue, const double Factor)
{
return (AValue / Factor);
}
```

The problem is, this approach requires extra parameters on the conversion function, which means you can't simply register the same function with every European currency. In order to avoid having to write two new conversion functions for every European currency, you can make use of the same two functions by making them the members of a class.

Creating the conversion class

The class must be a descendant of *TConvTypeFactor*. *TConvTypeFactor* defines two methods, *ToCommon* and *FromCommon*, for converting to and from the base units of a conversion family (in this case, to and from Euros). Just as with the functions you use directly when registering a conversion unit, these methods have no extra parameters, so you must supply the number of digits to round off and the conversion factor as private members of your conversion class:

```
type
TConvTypeEuroFactor = class(TConvTypeFactor)
private
FRound: TRoundToRange;
public
constructor Create(const AConvFamily: TConvFamily;
const ADescription: string; const AFactor: Double;
const ARound: TRoundToRange);
function ToCommon(const AValue: Double): Double; override;
function FromCommon(const AValue: Double): Double; override;
end;
end;
class PASCALIMPLEMENTATION TConvTypeEuroFactor : public Convutils::TConvTypeFactor
{
  private:
TRoundToRange FRound;
public:
    __fastcall TConvTypeEuroFactor(const TConvFamily AConvFamily,
```

```
        const AnsiString ADescription, const double AFactor, const TRoundToRange ARound);
              TConvTypeFactor(AConvFamily, ADescription, AFactor);
    virtual double ToCommon(const double AValue);
virtual double FromCommon(const double AValue);
}
```

The constructor assigns values to those private members:

```
constructor TConvTypeEuroFactor.Create(const AConvFamily: TConvFamily;
const ADescription: string; const AFactor: Double;
const ARound: TRoundToRange);
begin
inherited Create(AConvFamily, ADescription, AFactor);
FRound := ARound;
end;
__fastcall TConvTypeEuroFactor::TConvTypeEuroFactor(const TConvFamily AConvFamily,
        const AnsiString ADescription, const double AFactor, const TRoundToRange ARound):
              TConvTypeFactor(AConvFamily, ADescription, AFactor);
{
  FRound = ARound;
}
```

The two conversion functions simply use these private members:

```
function TConvTypeEuroFactor.FromCommon(const AValue: Double): Double;
begin
Result := RoundTo(AValue * Factor, FRound);
end;
function TConvTypeEuroFactor.ToCommon(const AValue: Double): Double;
begin
Result := AValue / Factor;
end;
virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return (RoundTo(AValue * Factor, FRound));
}
virtual double TConvTypeEuroFactor::ToCommon(const double AValue)
{
  return (AValue / Factor);
}
```

Declare variables

Now that you have a conversion class, begin as with any other conversion family, by declaring identifiers:

```
var
  euEUR: TConvType; { EU euro }
euBEF: TConvType; { Belgian francs }
euDEM: TConvType; { German marks }
euGRD: TConvType; { Greek drachmas }
euESP: TConvType; { Spanish pesetas }
euFFR: TConvType; { French francs }
euIEP: TConvType; { Irish pounds }
euITL: TConvType; { Italian lire }
euLUF: TConvType; { Luxembourg francs }
euNLG: TConvType; { Dutch guilders }
euATS: TConvType; { Austrian schillings }
euPTE: TConvType; { Portuguese escudos }
euFIM: TConvType; { Finnish marks }
  cbEuro: TConvFamily;
TConvFamily cbEuro;
TConvType euEUR; // EU euro
TConvType euBEF; // Belgian francs
TConvType euDEM; // German marks
TConvType euGRD; // Greek drachmas
TConvType euESP; // Spanish pesetas
TConvType euFFR; // French francs
TConvType euIEP; // Irish pounds
```

```
TConvType euITL; // Italian lire
TConvType euLUF; // Luxembourg francs
TConvType euNLG; // Dutch guilders
TConvType euATS; // Austrian schillings
TConvType euPTE; // Protuguese escudos
TConvType euFIM; // Finnish marks
```

Register the conversion family and the other units

Now you are ready to register the conversion family and the European monetary units, using your new conversion class. Register the conversion family the same way you registered the other conversion families:

```
cbEuro := RegisterConversionFamily ('European currency');
cbEuro = RegisterConversionFamily ("European currency");
```

To register each conversion type, create an instance of the conversion class that reflects the factor and rounding properties of that currency, and call the RegisterConversionType method:

```
var
  LInfo: TConvTypeInfo;
begin
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'EUEuro', 1.0, -2);
  if not RegisterConversionType(LInfo, euEUR) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'BelgianFrancs', 40.3399, 0);
  if not RegisterConversionType(LInfo, euBEF) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'GermanMarks', 1.95583, -2);
  if not RegisterConversionType(LInfo, euDEM) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'GreekDrachmas', 340.75, 0);
  if not RegisterConversionType(LInfo, euGRD) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'SpanishPesetas', 166.386, 0);
  if not RegisterConversionType(LInfo, euESP) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'FrenchFrancs', 6.55957, -2);
  if not RegisterConversionType(LInfo, euFFR) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'IrishPounds', 0.787564, -2);
  if not RegisterConversionType(LInfo, euIEP) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'ItalianLire', 1936.27, 0);
  if not RegisterConversionType(LInfo, euITL) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'LuxembourgFrancs', 40.3399, -2);
  if not RegisterConversionType(LInfo, euLUF) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'DutchGuilders', 2.20371, -2);
  if not RegisterConversionType(LInfo, euNLG) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'AustrianSchillings', 13.7603, -2);
  if not RegisterConversionType(LInfo, euATS) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'PortugueseEscudos', 200.482, -2);
  if not RegisterConversionType(LInfo, euPTE) then
    LInfo.Free;
  LInfo := TConvTypeEuroFactor.Create(cbEuro, 'FinnishMarks', 5.94573, 0);
  if not RegisterConversionType(LInfo, euFIM) then
    LInfo.Free;
end;
TConvTypeInfo *pInfo = new TConvTypeEuroFactor(cbEuro, "EUEuro", 1.0, -2);
if (!RegisterConversionType(pInfo, euEUR))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "BelgianFrancs", 40.3399, 0);
if (!RegisterConversionType(pInfo, euBEF))
  delete pInfo;
```

```
pInfo = new TConvTypeEuroFactor(cbEuro, "GermanMarks", 1.95583, -2);
if (!RegisterConversionType(pInfo, euDEM))
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "GreekDrachmas", 340.75, 0);
if (!RegisterConversionType(pInfo, euGRD)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "SpanishPesetas", 166.386, 0);
if (!RegisterConversionType(pInfo, euESP)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "FrenchFrancs", 6.55957, -2);
if (!RegisterConversionType(pInfo, euFFR)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "IrishPounds", 0.787564, -2);
if (!RegisterConversionType(pInfo, euIEP)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "ItalianLire", 1936.27, 0);
if (!RegisterConversionType(pInfo, euITL)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "LuxembourgFrancs", 40.3399, -2);
if (!RegisterConversionType(pInfo, euLUF)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "DutchGuilders", 2.20371, -2);
if (!RegisterConversionType(pInfo, euNLG)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "AutstrianSchillings", 13.7603, -2);
if (!RegisterConversionType(pInfo, euATS)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "PortugueseEscudos", 200.482, -2);
if (!RegisterConversionType(pInfo, euPTE)
  delete pInfo;
pInfo = new TConvTypeEuroFactor(cbEuro, "FinnishMarks", 5.94573, 0);
if (!RegisterConversionType(pInfo, euFIM)
  delete pInfo;
```

Note: The ConvertIt demo provides an expanded version of this example that includes other currencies (that do not have fixed conversion rates) and more error checking.

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the European currencies you have registered with the new cbEuro family. For example, the following code converts a value from Italian Lire to German Marks:

```
Edit2.Text = FloatToStr(Convert(StrToFloat(Edit1.Text), euITL, euDEM));
Edit2->Text = FloatToStr(Convert(StrToFloat(Edit1->Text), euITL, euDEM));
```

See Also

Performing Conversions (see page 2105)

3.2.4.10.27 Using a Conversion Function

For cases when the conversion is more complex, you can use a different syntax to specify a function to perform the conversion instead of using a conversion factor. For example, you can't convert temperature values using a conversion factor, because different temperature scales have a different origins.

This example, which comes from the StdConvs unit, shows how to register a conversion type by providing functions to convert to and from the base units.

Declare variables

First, declare variables for the identifiers. The identifiers are used in the *cbTemperature* conversion family, and the units of measurement are its members:

```
var
cbTemperature: TConvFamily;
tuCelsius: TConvType;
tuKelvin: TConvType;
tuFahrenheit: TConvType;
TConvFamily cbTemperature;
TConvType tuCelsius;
TConvType tuKelvin;
TConvType tuFahrenheit;
```

Note: The units of measurement listed here are a subset of the temperature units actually registered in the *StdConvs* unit.

Register the conversion family

Next, register the conversion family:

```
cbTemperature := RegisterConversionFamily ('Temperature');
cbTemperature = RegisterConversionFamily ("Temperature");
```

Register the base unit

Next, define and register the base unit of the conversion family, which in the example is degrees Celsius. Note that in the case of the base unit, we can use a simple conversion factor, because there is no actual conversion to make:

```
tuCelsius:=RegisterConversionType(cbTemperature,'Celsius',1);
tuCelsius = RegisterConversionType(cbTemperature,"Celsius",1);
```

Write methods to convert to and from the base unit

You need to write the code that performs the conversion from each temperature scale to and from degrees Celsius, because these do not rely on a simple conversion factor. These functions are taken from the StdConvs unit:

```
function FahrenheitToCelsius(const AValue: Double): Double;
begin
Result := ((AValue - 32) * 5) / 9;
end;
function CelsiusToFahrenheit(const AValue: Double): Double;
begin
Result := ((AValue * 9) / 5) + 32;
end;
function KelvinToCelsius(const AValue: Double): Double;
begin
Result := AValue - 273.15;
end;
function CelsiusToKelvin(const AValue: Double): Double;
begin
Result := AValue + 273.15;
end;
double __fastcall FahrenheitToCelsius(const double AValue)
{
return (((AValue - 32) * 5) / 9);
}
double __fastcall CelsiusToFahrenheit(const double AValue)
{
return (((AValue * 9) / 5) + 32);
}
double __fastcall KelvinToCelsius(const double AValue)
{
  return (AValue - 273.15);
}
double __fastcall CelsiusToKelvin(const double AValue)
{
  return (AValue + 273.15);
}
```

Register the other units

Now that you have the conversion functions, you can register the other measurement units within the conversion family. You also

include a description of the units.

The code to register the other units in the family is shown here:

```
tuKelvin := RegisterConversionType(cbTemperature, 'Kelvin', KelvinToCelsius, CelsiusToKelvin);
tuFahrenheit := RegisterConversionType(cbTemperature, 'Fahrenheit', FahrenheitToCelsius,
CelsiusToFahrenheit);
tuKelvin = RegisterConversionType(cbTemperature, "Kelvin", KelvinToCelsius, CelsiusToKelvin);
tuFahrenheit = RegisterConversionType(cbTemperature, "Fahrenheit", FahrenheitToCelsius,
CelsiusToFahrenheit);
```

Use the new units

You can now use the newly registered units to perform conversions in your applications. The global *Convert* function can convert between any of the conversion types that you registered with the *cbTemperature* conversion family. For example the following code converts a value from degrees Fahrenheit to degrees Kelvin.

```
Convert(StrToFloat(Edit1.Text), tuFahrenheit, tuKelvin);
Convert(StrToFloat(Edit1->Text), tuFahrenheit, tuKelvin);
```

See Also

Performing Conversions (see page 2105)

3.2.4.10.28 Using Streams to Read or Write Data

Stream classes all share several methods for reading and writing data. These methods are distinguished by whether they:

- Return the number of bytes read or written.
- Require the number of bytes to be known.
- Raise an exception on error.

Stream methods for reading and writing

The *Read* method reads a specified number of bytes from the stream, starting at its current *Position*, into a buffer. *Read* then advances the current position by the number of bytes actually transferred. The prototype for *Read* is:

```
function Read(var Buffer; Count: Longint): Longint;
virtual int __fastcall Read(void *Buffer, int Count);
```

Read is useful when the number of bytes in the file is not known. *Read* returns the number of bytes actually transferred, which may be less than *Count* if the stream did not contain *Count* bytes of data past the current position.

The *Write* method writes *Count* bytes from a buffer to the stream, starting at the current *Position*. The prototype for *Write* is:

```
function Write(const Buffer; Count: Longint): Longint;
virtual int __fastcall Write(const void *Buffer, int Count);
```

After writing to the file, *Write* advances the current position by the number bytes written, and returns the number of bytes actually written, which may be less than *Count* if the end of the buffer is encountered or the stream can't accept any more bytes.

The counterpart procedures are *ReadBuffer* and *WriteBuffer* which, unlike *Read* and *Write*, do not return the number of bytes read or written. These procedures are useful in cases where the number of bytes is known and required, for example when reading in structures. *ReadBuffer* and *WriteBuffer* raise an exception (*EReadError* and *EWriteError*) if the byte count can not be matched exactly. This is in contrast to the *Read* and *Write* methods, which can return a byte count that differs from the requested value. The prototypes for *ReadBuffer* and *WriteBuffer* are:

```
procedure ReadBuffer(var Buffer; Count: Longint);
procedure WriteBuffer(const Buffer; Count: Longint);
virtual int __fastcall ReadBuffer(void *Buffer, int Count);
virtual int __fastcall WriteBuffer(const void *Buffer, int Count);
```

These methods call the *Read* and *Write* methods to perform the actual reading and writing.

Reading and writing components

TStream defines specialized methods, ReadComponent and WriteComponent, for reading and writing components. You can use them in your applications as a way to save components and their properties when you create or alter them at runtime.

ReadComponent and *WriteComponent* are the methods that the IDE uses to read components from or write them to form files. When streaming components to or from a form file, stream classes work with the *TFiler* classes, *TReader* and *TWriter*, to read objects from the form file or write them out to disk. For more information about using the component streaming system, see TStream, TFiler, TReader, TWriter, and TComponent classes.

Reading and writing strings

If you are passing a string to a read or write function, you need to be aware of the correct syntax. The *Buffer* parameters for the read and write routines are **var** and **const** types, respectively. These are untyped parameters, so the routine takes the address of a variable.

The most commonly used type when working with strings is a long string. However, passing a long string as the *Buffer* parameter does not produce the correct result. Long strings contain a size, a reference count, and a pointer to the characters in the string. Consequently, dereferencing a long string does not result in the pointer element. You need to first cast the string to a *Pointer* or *PChar*, and then dereference it. For example:

```
procedure caststring;
var
  fs: TFileStream;
const
  s: string = 'Hello';
begin
  fs := TFileStream.Create('temp.txt', fmCreate or fmOpenWrite);
  fs.Write(s, Length(s));// this will give you garbage
  fs.Write(PChar(s)^, Length(s));// this is the correct way
end;
```

See Also

Copying Data from One Stream to Another (see page 2094)

Specifying the Stream Position and Size (see page 2106)

3.2.4.10.29 Using the TCustomVariantType Descendant

In the initialization section of the unit that defines your *TCustomVariantType* descendant, create an instance of your class. When you instantiate your object, it automatically registers itself with the variant-handling system so that the new Variant type is enabled. For example, here is the initialization section of the VarCmplx unit:

```
initialization
ComplexVariantType := TComplexVariantType.Create;
```

In the finalization section of the unit that defines your *TCustomVariantType* descendant, free the instance of your class. This automatically unregisters the variant type. Here is the finalization section of the VarCmplx unit:

```
finalization
FreeAndNil(ComplexVariantType);
```

See Also

Implementing Binary Operations (see page 2100)

Implementing Comparison Operations (see page 2102)

Implementing Unary Operations (see page 2103)

Enabling Casting (see page 2097)

Copying and Clearing Custom Variants (see page 2093)

Loading and Saving Custom Variant Values (see page 2104)

3.2.4.10.30 Writing Utilities to Work with a Custom Variant Type

Once you have created a *TCustomVariantType* descendant to implement your custom variant type, it is possible to use the new Variant type in applications. However, without a few utilities, this is not as easy as it should be.

It is a good idea to create a method that creates an instance of your custom variant type from an appropriate value or set of values. This function or set of functions fills out the structure you defined (see page 2107) to store your custom variant's data. For example, the following function could be used to create a complex-valued variant:

```
function VarComplexCreate(const AReal, AImaginary: Double): Variant;
begin
  VarClear(Result);
TComplexVarData(Result).VType := ComplexVariantType.VarType;
TComplexVarData(ADest).VComplex := TComplexData.Create(ARead, AImaginary);
end;
```

This function does not actually exist in the VarCmplx unit, but is a synthesis of methods that do exist, provided to simplify the example. Note that the returned variant is cast to the record that was defined to map onto the *TVarData* structure (*TComplexVarData*), and then filled out.

Another useful utility to create is one that returns the variant type code for your new Variant type. This type code is not a constant. It is automatically generated when you instantiate your *TCustomVariantType* descendant. It is therefore useful to provide a way to easily determine the type code for your custom variant type. The following function from the VarCmplx unit illustrates how to write one, by simply returning the *VarType* property of the *TCustomVariantType* descendant:

```
function VarComplex: TVarType;
begin
Result := ComplexVariantType.VarType;
end;
```

Two other standard utilities provided for most custom variants check whether a given variant is of the custom type and cast an arbitrary variant to the new custom type. Here is the implementation of those utilities from the VarCmplx unit:

```
function VarIsComplex(const AValue: Variant): Boolean;
begin
Result := (TVarData(AValue).VType and varTypeMask) = VarComplex;
end;
function VarAsComplex(const AValue: Variant): Variant;
begin
if not VarIsComplex(AValue) then
VarCast(Result, AValue, VarComplex)
else
Result := AValue;
end;
```

Note that these use standard features of all variants: the *VType* member of the *TVarData* record and the *VarCast* function, which works because of the methods implemented in the TCustomVariantType descendant for casting data (see page 2097).

In addition to the standard utilities mentioned above, you can write any number of utilities specific to your new custom variant type. For example, the VarCmplx unit defines a large number of functions that implement mathematical operations on complex-valued variants.

See Also

Storing a Custom Variant Type's Data (see page 2107)

Creating a Class to Enable the Custom Variant Type (see page 2094)

Supporting Properties and Methods in Custom Variants (see page 2108)

3.2.4.10.31 Commonly Used Long String Routines

The string handling routines in `SysUtils` handle both **UnicodeString** and **AnsiString** types. Within their functional areas, some routines are used for the same purpose, the differences being whether they use a particular criterion in their calculations.

The `AnsiStrings` unit's functions offer the same capabilities as the `SysUtils` unit's routines. `AnsiStrings` functions provide better performance for **AnsiString** than `SysUtils` functions, because `SysUtils` functions convert an **AnsiString** to a **UnicodeString**. `AnsiStrings` functions work only with **AnsiString**, so there is no conversion.

The following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string

Where appropriate, the tables also provide columns indicating whether a routine satisfies the following criteria.

- Uses case sensitivity: If locale settings are used, it determines the definition of case. If the routine does not use locale settings, analyses are based upon the ordinal values of the characters. If the routine is case-insensitive, there is a logical merging of upper and lower case characters that is determined by a predefined pattern.
- Uses locale settings: Locale settings allow you to customize your application for specific locales, in particular, for Asian language environments. Most locale settings consider lowercase characters to be less than the corresponding uppercase characters. This is in contrast to ASCII order, in which lowercase characters are greater than uppercase characters. Routines that use the system locale are typically prefaced with Ansi (that is, AnsiXXX).
- Supports the multi-byte character set (MBCS): MBCSs are used when writing code for far eastern locales. Multi-byte characters are represented by one or more character codes, so the length in bytes does not necessarily correspond to the length of the string. The routines that support MBCS parse one- and multibyte characters.

ByteType and *StrByteType* determine whether a particular byte is the lead byte of a multibyte character. Be careful when using multibyte characters not to truncate a string by cutting a character in half. Do not pass characters as a parameter to a function or procedure, since the size of a character cannot be predetermined. Pass, instead, a pointer to a to a character or string. For more information about MBCS, see Enabling Application Code (see page 1924).

String length routines:

Routine	Uses locale settings	Supports MBCS	Returns
System.Length	yes	yes	Number of *elements* in string. Does not account for MBCS or Unicode surrogate pair "characters".
System.SizeOf	yes	yes	Number of *bytes* required to represent type or variable.

Note that `Length` and `SizeOf` give different results for all strings except for short strings. `SizeOf` equals the number of bytes in a pointer, since non-short strings are pointers.

String comparison routines:

Routine	Case-sensitive	Uses locale settings	Supports MBCS
AnsiStrings.AnsiCompareStr SysUtils.AnsiCompareStr	yes	yes	yes
AnsiStrings.AnsiCompareText SysUtils.AnsiCompareText	no	yes	yes

AnsiStrings.AnsiCompareFileName SysUtils.AnsiCompareFileName	no	yes	yes
AnsiStrings.AnsiMatchStr StrUtils.AnsiMatchStr	yes	yes	yes
AnsiStrings.AnsiMatchText StrUtils.AnsiMatchText	no	yes	yes
AnsiStrings.AnsiContainsStr StrUtils.AnsiContainsStr	yes	yes	yes
AnsiStrings.AnsiContainsText StrUtils.AnsiContainsText	no	yes	yes
AnsiStrings.AnsiStartsStr StrUtils.AnsiStartsStr	yes	yes	yes
AnsiStrings.AnsiStartsText StrUtils.AnsiStartsText	no	yes	yes
AnsiStrings.AnsiEndsStr StrUtils.AnsiEndsStr	yes	yes	yes
AnsiStrings.AnsiEndsText StrUtils.AnsiEndsText	no	yes	yes
AnsiStrings.AnsiEndsStr StrUtils.AnsiEndsStr	yes	yes	yes
AnsiStrings.AnsiIndexText StrUtils.AnsiIndexText	no	yes	yes
AnsiStrings.CompareStr SysUtils.CompareStr	yes	no	no
AnsiStrings.CompareText SysUtils.CompareText	no	no	no
StrUtils.AnsiResemblesText	no	no	no

Case conversion routines:

Routine	Uses locale settings	Supports MBCS
AnsiStrings.AnsiLowerCase SysUtils.AnsiLowerCase	yes	yes
AnsiStrings.AnsiUpperCaseFileName SysUtils.AnsiUpperCaseFileName	yes	yes
AnsiStrings.AnsiUpperCase SysUtils.AnsiUpperCase	yes	yes
AnsiStrings.LowerCase SysUtils.LowerCase	no	no. Works for A to Z only, not European accented characters.
AnsiStrings.UpperCase SysUtils.UpperCase	no	no. Works for a to z only, not European accented characters.

Note: The routines used for string file names: *AnsiCompareFileName*, *AnsiLowerCaseFileName*, and *AnsiUpperCaseFileName* all use the system locale. You should always use file names that are portable because the locale (character set) used for file names can and might differ from the default user interface.

String modification routines:

Routine	Case-sensitive	Supports MBCS
SysUtils.AdjustLineBreaks	NA	yes
AnsiStrings.AnsiQuotedStr SysUtils.AnsiQuotedStr	NA	yes
AnsiStrings.AnsiReplaceStr StrUtils.AnsiReplaceStr	yes	yes
AnsiStrings.AnsiReplaceText StrUtils.AnsiReplaceText	no	yes
AnsiStrings.StringReplace SysUtils.StringReplace	optional by flag	yes
AnsiStrings.ReverseString StrUtils.ReverseString	NA	no
AnsiStrings.StuffString StrUtils.StuffString	NA	no
System.FillChar	NA	no
SysUtils.StrPCopy	yes	yes
AnsiStrings.Trim SysUtils.Trim	NA	yes
AnsiStrings.TrimLeft SysUtils.TrimLeft	NA	yes
AnsiStrings.TrimRight SysUtils.TrimRight	NA	yes
SysUtils.WrapText	NA	yes

Sub-string routines:

Routine	Case-sensitive	Supports MBCS
SysUtils.AnsiExtractQuotedStr	NA	yes
AnsiStrings.AnsiPos SysUtils.AnsiPos	yes	yes
AnsiStrings.IsDelimiter SysUtils.IsDelimiter	yes	yes
AnsiStrings.IsPathDelimiter SysUtils.IsPathDelimiter	yes	yes
AnsiStrings.LastDelimiter SysUtils.LastDelimiter	yes	yes
StrUtils.LeftStr	NA	no

StrUtils.RightStr	NA	no
StrUtils.MidStr	NA	no
AnsiStrings.QuotedStr SysUtils.QuotedStr	no	no
System.StringOfChar	NA	yes

String information routines (in System):

Routine	Returns
System.StringCodePage	Code page
System.StringElementSize	Element size
System.StringRefCount	Reference count

Note: It is not safe to directly manipulate the internal structures of strings, since these change with product versions.

See Also

Wide Character Routines (see page 2141)

Commonly Used Routines for Null-terminated Strings (see page 2121)

Enabling Unicode in Your Applications

Unicode in RAD Studio

3.2.4.10.32 Commonly Used Routines for Null-terminated Strings

The null-terminated string handling routines cover several functional areas. Within these areas, some are used for the same purpose, the differences being whether or not they use a particular criteria in their calculations. The following tables list these routines by these functional areas:

- Comparison
- Case conversion
- Modification
- Sub-string
- Copying

Where appropriate, the tables also provide columns indicating whether the routine is case-sensitive, uses the current locale, and/or supports multi-byte character sets.

Null-terminated string comparison routines

Routine	Case-sensitive	Uses locale settings	Supports MBCS
SysUtils.AnsiStrComp	yes	yes	yes
SysUtils.AnsiStrIComp	no	yes	yes
SysUtils.AnsiStrLComp	yes	yes	yes
SysUtils.AnsiStrLIComp	no	yes	yes
SysUtils.StrComp	yes	no	no
SysUtils.StrIComp	no	no	no
SysUtils.StrLComp	yes	no	no

SysUtils.StrLIComp	no	no	no

Null-terminated case conversion routines

Routine	Uses locale settings	Supports MBCS
SysUtils.AnsiStrLower	yes	yes
SysUtils.AnsiStrUpper	yes	yes
SysUtils.StrLower	no	no
SysUtils.StrUpper	no	no

String modification routines

Routine
SysUtils.StrCat
SysUtils.StrLCat

Sub-string routines

Routine	Case-sensitive	Supports MBCS
SysUtils.AnsiStrPos	yes	yes
SysUtils.AnsiStrScan	yes	yes
SysUtils.AnsiStrRScan	yes	yes
SysUtils.StrPos	yes	no
SysUtils.StrScan	yes	no
SysUtils.StrRScan	yes	no

Null-terminated string copying

Routine
SysUtils.StrCopy
SysUtils.StrLCopy
SysUtils.StrECopy
SysUtils.StrMove
SysUtils.StrPCopy
SysUtils.StrPLCopy

See Also

Commonly Used Long String Routines (see page 2118)

3.2.4.10.33 Compiler Directives for Strings

The following compiler directives affect character and string types.

Compiler directives for strings

Directive	Description
{$H+/-}	A compiler directive, $H, controls whether the reserved word **string** represents a short string or a long string. In the default state, {$H+}, **string** represents a long string. You can change it to a *ShortString* by using the {$H-} directive.
{$P+/-}	The $P directive is meaningful only for code compiled in the {$H-} state, and is provided for backwards compatibility. $P controls the meaning of variable parameters declared using the string keyword in the {$H-} state. In the {$P-} state, variable parameters declared using the string keyword are normal variable parameters, but in the {$P+} state, they are open string parameters. Regardless of the setting of the $P directive, the OpenString identifier can always be used to declare open string parameters.
{$V+/-}	The $V directive controls type checking on short strings passed as variable parameters. In the {$V+} state, strict type checking is performed, requiring the formal and actual parameters to be of identical string types. In the {$V-} (relaxed) state, any short string type variable is allowed as an actual parameter, even if the declared maximum length is not the same as that of the formal parameter. Be aware that this could lead to memory corruption. For example: var S: string[3]; procedure Test(var T: string); begin T := '1234'; end; begin Test(S); end.
{$X+/-}	The {$X+} compiler directive enables support for null-terminated strings by activating the special rules that apply to the built-in PChar type and zero-based character arrays. (These rules allow zero-based arrays and character pointers to be used with *Write*, *Writeln*, *Val*, *Assign*, and *Rename* from the System unit.)

See Also

Declaring and Initializing Strings (see page 2123)

3.2.4.10.34 Copying a File

The runtime library does not provide any routines for copying a file. However, if you are writing Windows-only applications, you can directly call the Windows API *CopyFile* function to copy a file. Like most of the runtime library file routines, *CopyFile* takes a filename as a parameter, not a file handle. When copying a file, be aware that the file attributes for the existing file are copied to the new file, but the security attributes are not. *CopyFile* is also useful when moving files across drives because neither the *RenameFile* function nor the Windows API *MoveFile* function can rename or move files across drives.

See Also

Finding a File (see page 2125)

Renaming a File (see page 2130)

File Date-time Routines (see page 2125)

Deleting a File (see page 2124)

3.2.4.10.35 Declaring and Initializing Strings

When you declare a string:

```
S: string;
```

you do not need to initialize it. Strings are automatically initialized to empty. To test a string for empty you can either use the *EmptyStr* variable:

```
S = EmptyStr;
```

or test against an empty string:

```
S = '';
```

An empty string has no valid data. Therefore, trying to index an empty string is like trying to access **nil** and will result in an access violation:

```
var
  S: string;
begin
  S[i];    // this will cause an access violation
  // statements
end;
```

Similarly, if you cast an empty string to a *PChar*, the result is a **nil** pointer. So, if you are passing such a *PChar* to a routine that needs to read or write to it, be sure that the routine can handle **nil**:

```
var
  S: string;   // empty string
begin
  proc(PChar(S));  // be sure that proc can handle nil
  // statements
end;
```

If it cannot, then you can either initialize the string:

```
  S := 'No longer nil';
  proc(PChar(S));// proc does not need to handle nil now
```

or set the length, using the *SetLength* procedure:

```
  SetLength(S, 100);//sets the dynamic length of S to 100 characters
  proc(PChar(S));// proc does not need to handle nil now
```

When you use *SetLength*, existing characters in the string are preserved, but the contents of any newly allocated space is undefined. Following a call to *SetLength*, S is guaranteed to reference a unique string, that is a string with a reference count of one.

String literals are Unicode by default. To initialize a string to ANSI characters, cast the literal:

```
AnsiString S;
S := AnsiString('Unicode string');
```

To obtain the length of a string in *elements*, use the `Length` function. Note that for MBCS and Unicode strings, this is not necessarily the number of characters.

To obtain the length of a short string in *bytes*, use the `SizeOf` function. For the other string types, multiply `Length` by `SizeOf`. For instance, if S is a **UnicodeString**, its length in bytes is `Length(S) * SizeOf(S)`.

Remember when declaring a **string** that:

```
  S: string[n];
```

implicitly declares a short string, not a string of *n* length. To declare a string of specifically *n* length, declare a variable of type **string** and use the *SetLength* procedure.

```
  S: string;
  SetLength(S, n);
```

See Also

Enabling Unicode in Your Applications

Mixing and Converting String Types (see page 2127)

Unicode in RAD Studio

3.2.4.10.36 Deleting a File

Deleting a file erases the file from the disk and removes the entry from the disk's directory. There is no corresponding operation to restore a deleted file, so applications should generally allow users to confirm before deleting files. To delete a file, pass the name of the file to the *DeleteFile* function:

```
DeleteFile(FileName);
```

DeleteFile returns *True* if it deleted the file and *False* if it did not (for example, if the file did not exist or if it was read-only). *DeleteFile* erases the file named by *FileName* from the disk.

See Also

Finding a File (see page 2125)

Renaming a File (see page 2130)

File Date-time Routines (see page 2125)

Copying a File (see page 2123)

3.2.4.10.37 File Date-time Routines

The *FileAge*, *FileGetDate*, and *FileSetDate* routines operate on operating system date-time values. *FileAge* returns the date-and-time stamp of a file, or -1 if the file does not exist. *FileSetDate* sets the date-and-time stamp for a specified file, and returns zero on success or an error code on failure. *FileGetDate* returns a date-and-time stamp for the specified file or –1 if the handle is invalid.

As with most of the file manipulating routines, *FileAge* uses a string filename. *FileGetDate* and *FileSetDate*, however, use a *Handle* type as a parameter. To get the file handle either:

- Use the *FileOpen* or *FileCreate* function to create a new file or open an existing file. Both *FileOpen* and *FileCreate* return the file handle.
- Instantiate *TFileStream* to create or open a file. Then use its *Handle* property. See Using file streams (see page 2133) for more information.

See Also

Finding a File (see page 2125)

Renaming a File (see page 2130)

Deleting a File (see page 2124)

Copying a File (see page 2123)

3.2.4.10.38 Finding a File

There are three routines used for finding a file: *FindFirst*, *FindNext*, and *FindClose*. *FindFirst* searches for the first instance of a filename with a given set of attributes in a specified directory. *FindNext* returns the next entry matching the name and attributes specified in a previous call to *FindFirst*. *FindClose* releases memory allocated by *FindFirst*. You should always use *FindClose* to terminate a *FindFirst*/*FindNext* sequence. If you want to know if a file exists, a *FileExists* function returns *True* if the file exists, *False* otherwise.

The three file find routines take a *TSearchRec* as one of the parameters. *TSearchRec* defines the file information searched for by *FindFirst* or *FindNext*. If a file is found, the fields of the *TSearchRec* type parameter are modified to describe the found file.

```
type
  TFileName = string;
  TSearchRec = record
    Time: Integer;//Time contains the time stamp of the file.
    Size: Integer;//Size contains the size of the file in bytes.
    Attr: Integer;//Attr represents the file attributes of the file.
    Name: TFileName;//Name contains the filename and extension.
    ExcludeAttr: Integer;
    FindHandle: THandle;
    FindData: TWin32FindData;//FindData contains additional information such as
```

```
    //file creation time, last access time, long and short filenames.
  end;
struct TSearchRec
{
int Time; // time stamp of the file
int Size; // size of the file in bytes
int Attr; // file attribute flags
AnsiString Name; // filename and extension
int ExcludeAttr; // file attribute flags for files to ignore
unsigned FindHandle;
_WIN32_FIND_DATAA FindData; // structure with addition information
} ;
```

On field of *TSearchRec* that is of particular interest is the *Attr* field. You can test *Attr* against the following attribute constants or values to determine if a file has a specific attribute:

Attribute constants and values

Constant	Value	Description
faReadOnly	$00000001	Read-only files
faHidden	$00000002	Hidden files
faSysFile	$00000004	System files
faVolumeID	$00000008	Volume ID files
faDirectory	$00000010	Directory files
faArchive	$00000020	Archive files
faAnyFile	$0000003F	Any file

To test for an attribute, combine the value of the *Attr* field with the attribute constant using the **and** operator. If the file has that attribute, the result will be greater than 0. For example, if the found file is a hidden file, the following expression will evaluate to *True*:

```
(SearchRec.Attr and faHidden > 0).
(SearchRec.Attr & faHidden > 0).
```

Attributes can be combined by OR'ing their constants or values. For example, to search for read-only and hidden files in addition to normal files, pass the following as the *Attr* parameter.

```
(faReadOnly or faHidden).
(faReadOnly | faHidden).
```

The following example illustrates the use of the three file find routines. It uses a label, a button named *Search*, and a button named *Again* on a form. When the user clicks the *Search* button, the first file in the specified path is found, and the name and the number of bytes in the file appear in the label's caption. Each time the user clicks the *Again* button, the next matching filename and size is displayed in the label:

```
var
  SearchRec: TSearchRec;
procedure TForm1.SearchClick(Sender: TObject);
begin
  FindFirst('c:\Program Files\MyProgram\bin\

*.*', faAnyFile, SearchRec);
  Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + " bytes in size';
end;
procedure TForm1.AgainClick(Sender: TObject);
begin
  if FindNext(SearchRec) = 0 then
    Label1.Caption := SearchRec.Name + ' is ' + IntToStr(SearchRec.Size) + ' bytes in size'
  else
    FindClose(SearchRec);
```

```
end;
  TSearchRec SearchRec; // global variable
void __fastcall TForm1::SearchClick(TObject *Sender)
{
  FindFirst("c:\\Program Files\\MyProgram\\bin\\
*.*", faAnyFile, SearchRec);
  Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size) + " bytes in size";
}
void __fastcall TForm1::AgainClick(TObject *Sender)
{
  if (FindNext(SearchRec) == 0)
    Label1->Caption = SearchRec->Name + " is " + IntToStr(SearchRec.Size) + " bytes in size";
  else
    FindClose(SearchRec);
}
```

Note: In cross-platform applications, you should replace any hard-coded pathnames with the correct pathname for the system or use environment variables (on the Environment Variables page when you choose Tools->Options->Environment Options

) to represent them.

See Also

Deleting a File (see page 2124)

Renaming a File (see page 2130)

File Date-time Routines (see page 2125)

Copying a File (see page 2123)

3.2.4.10.39 Manipulating Files

Several common file operations are built into the runtime library. The routines for working with files operate at a high level. For most routines, you specify the name of the file and the routine makes the necessary calls to the operating system for you. In some cases, you use file handles instead.

Warning: Although the Delphi language is not case sensitive, the Linux operating system is. Be attentive to case when working with files in cross-platform applications.

The following topics describe how to use runtime library routines to perform file manipulation tasks:

- Deleting a file (see page 2124)
- Finding a file (see page 2125)
- Renaming a file (see page 2130)
- File date-time routines (see page 2125)
- Copying a file (see page 2123)

3.2.4.10.40 Mixing and Converting String Types

Short, long, and wide strings can be mixed in assignments and expressions, and the compiler automatically generates code to perform the necessary string type conversions. However, when assigning a string value to a short string variable, be aware that the string value is truncated if it is longer than the declared maximum length of the short string variable.

Long strings are already dynamically allocated. If you use one of the built-in pointer types, such as *PAnsiString*, *PString*, or *PWideString*, remember that you are introducing another level of indirection. Be sure this is what you intend.

See Also

Declaring and Initializing Strings (see page 2123)

3.2.4.10.41 Passing a Local Variable as a PChar

Consider the case where you have a local string variable that you need to initialize by calling a function that takes a *PChar*. One approach is to create a local **array of char** and pass it to the function, then assign that variable to the string:

```
// assume FillBuffer is a predefined function
function FillBuffer(Value:Integer; Buf:PChar; Count:Integer):Integer
begin
  . . .
end;
// assume MAX_SIZE is a predefined constant
var
  i: Integer;
  buf: array[0..MAX_SIZE] of char;
  S: string;
begin
  i := FillBuffer(0, buf, SizeOf(buf));// treats buf as a PChar
  S := buf;
  //statements
end;
```

This approach is useful if the size of the buffer is relatively small, since it is allocated on the stack. It is also safe, since the conversion between an **array of char** and a **string** is automatic. The *Length* of the string is automatically set to the right value after assigning *buf* to the string.

To eliminate the overhead of copying the buffer, you can cast the string to a *PChar* (if you are certain that the routine does not need the *PChar* to remain in memory). However, synchronizing the length of the string does not happen automatically, as it does when you assign an **array of char** to a **string**. You should reset the string *Length* so that it reflects the actual width of the string. If you are using a function that returns the number of bytes copied, you can do this safely with one line of code:

```
var
  S: string;
begin
  SetLength(S, MAX_SIZE);// when casting to a PChar, be sure the string is not empty
  SetLength(S, GetModuleFilename(0, PChar(S), Length(S)));
  // statements
end;
```

See Also

String Dependencies (see page 2108)

Returning a PChar Local Variable (see page 2131)

3.2.4.10.42 Creating Drawing Spaces

The *TCanvas* class is defined in the Graphics unit, and encapsulates a Windows device context. This class handles all drawing for forms, visual containers (such as panels) and the printer object (see Printing (see page 2129)). Using the canvas object, you need not worry about allocating pens, brushes, palettes, and so on—all the allocation and deallocation are handled for you.

TCanvas includes a large number of primitive graphics routines to draw lines, shapes, polygons, fonts, etc. onto any control that contains a canvas. For example, here is a button event handler that draws a line from the upper left corner to the middle of the form and outputs some raw text onto the form:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Canvas.Pen.Color := clBlue;
Canvas.MoveTo( 10, 10 );
Canvas.LineTo( 100, 100 );
Canvas.Brush.Color := clBtnFace;
Canvas.Font.Name := 'Arial';
```

```
Canvas.TextOut( Canvas.PenPos.x, Canvas.PenPos.y,'This is the end of the line' );
end;
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Canvas->Pen->Color = clBlue;
Canvas->MoveTo( 10, 10 );
Canvas->LineTo( 100, 100 );
Canvas->Brush->Color = clBtnFace;
Canvas->Font->Name = "Arial";
Canvas->TextOut( Canvas->PenPos.x, Canvas->PenPos.y,"This is the end of the line" );
}
```

The *TCanvas* object defined in the Graphics unit also protects you against common Windows graphics errors, such as restoring device contexts, pens, brushes, and so on to the value they had before the drawing operation. *TCanvas* is used everywhere in the VCL that drawing is required or possible, and makes drawing graphics both fail-safe and easy.

See Also

Printing (see page 2129)

3.2.4.10.43 Printing

The VCL *TPrinter* object encapsulates details of Windows printers. To get a list of installed and available printers, use the *Printers* property. Both printer objects use a *TCanvas* (which is identical to the form's *TCanvas*) which means that anything that can be drawn on a form can be printed as well. To print an image, call the *BeginDoc* method followed by whatever canvas graphics you want to print (including text through the *TextOut* method) and send the job to the printer by calling the *EndDoc* method.

This example uses a button and a memo on a form. When the user clicks the button, the content of the memo is printed with a 200-pixel border around the page.

To run this example successfully, add Printers to your **uses** clause.

```
procedure TForm1.Button1Click(Sender: TObject);
var
r: TRect;
i: Integer;
begin
with Printer do
begin
r := Rect(200,200,(Pagewidth - 200),(PageHeight - 200));
BeginDoc;
      Canvas.Brush.Style := bsClear;
for i := 0 to Memo1.Lines.Count do
Canvas.TextOut(200,200 + (i *
                  Canvas.TextHeight(Memo1.Lines.Strings[i])),
                  Memo1.Lines.Strings[i]);
Canvas.Brush.Color := clBlack;
Canvas.FrameRect(r);
EndDoc;
end;
end;
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TPrinter *Prntr = Printer();
TRect r = Rect(200,200,Prntr->PageWidth - 200,Prntr->PageHeight- 200);
Prntr->BeginDoc();
for( int i = 0; i < Memo1->Lines->Count; i++)
Prntr->Canvas->TextOut(200,200 + (i *
  Prntr->Canvas->TextHeight(Memo1->Lines->Strings[i])),
  Memo1->Lines->Strings[i]);
Prntr->Canvas->Brush->Color = clBlack;
Prntr->Canvas->FrameRect(r);
```

```
Prntr->EndDoc();
}
```

3.2.4.10.44 Using TRegistryIniFile

Many 32-bit Windows applications store their information in the system Registry instead of ini files because the Registry is hierarchical and doesn't suffer from the size limitations of ini files. If you are accustomed to using ini files and want to move your configuration information to the Registry instead, you can use the *TRegistryIniFile* class. You may also want to use *TRegistryIniFile* in cross-platform applications if you want to use the system Registry on Windows and an ini file on Linux. You can write most of your application so that it uses the *TCustomIniFile* type. You need only conditionalize the code that creates an instance of *TRegistryIniFile* (on Windows) or *TMemIniFile* (on Linux) and assigns it to the *TCustomIniFile* your application uses.

TRegistryIniFile makes Registry entries look like ini file entries. All the methods from *TIniFile* and *TMemIniFile* (read and write) exist in *TRegistryIniFile*.

When you construct a *TRegistryIniFile* object, the parameter you pass to the constructor (corresponding to the filename for an *IniFile* or *TMemIniFile* object) becomes a key value under the user key in the registry. All sections and values branch from that root. *TRegistryIniFile* simplifies the Registry interface considerably, so you may want to use it instead of the *TRegistry* component even if you aren't porting existing code or writing a cross-platform application.

See Also

Using TINIFile (see page 2131)

Using TRegistry (see page 2099)

3.2.4.10.45 Renaming a File

To change a file name, use the *RenameFile* function:

```
function RenameFile(const OldFileName, NewFileName: string): Boolean;
extern PACKAGE bool __fastcall RenameFile(const AnsiString OldName, const AnsiString NewName);
```

RenameFile changes a file name, identified by *OldFileName*, to the name specified by *NewFileName*. If the operation succeeds, *RenameFile* returns *True*. If it cannot rename the file (for example, if a file called *NewFileName* already exists), *RenameFile* returns *False*. For example:

```
if not RenameFile('OLDNAME.TXT','NEWNAME.TXT') then
  ErrorMsg('Error renaming file!');
if (!RenameFile("OLDNAME.TXT","NEWNAME.TXT"))
  ErrorMsg("Error renaming file!");
```

You cannot rename (move) a file across drives using *RenameFile*. You would need to first copy the file and then delete the old one.

Note: *RenameFile* in the runtime library is a wrapper around the Windows API *MoveFile* function, so *MoveFile* will not work across drives either.

See Also

Finding a File (see page 2125)

Deleting a File (see page 2124)

File Date-time Routines (see page 2125)

Copying a File (see page 2123)

3.2.4.10.46 Returning a PChar Local Variable

A common error when working with *PChars* is to store a local variable in a data structure, or return it as a value. When your routine ends, the *PChar* disappears because it is a pointer to memory, and not a reference counted copy of the string. For example:

```
function title(n: Integer): PChar;
var
  s: string;
begin
  s := Format('title - %d', [n]);
  Result := PChar(s); // DON'T DO THIS
end;
```

This example returns a pointer to string data that is freed when the *title* function returns.

See Also

String Dependencies (see page 2108)

Passing a Local Variable as a PChar (see page 2128)

3.2.4.10.47 String to PChar Conversions

Long string (**AnsiString** and **UnicodeString**) to *PChar* conversions are not automatic. Some of the differences between strings and *PChars* can make conversions problematic:

- Long strings are reference-counted, while *PChars* are not.
- Assigning to a string copies the data, while a *PChar* is a pointer to memory.
- Long strings are null-terminated and also contain the length of the string and other information, while *PChars* are simply null-terminated.

Situations in which these differences can cause subtle errors are discussed in the following topics:

- String dependencies (see page 2108)
- Returning a PChar local variable (see page 2131)
- Passing a local variable as a PChar (see page 2128)

See Also

Declaring and Initializing Strings (see page 2123)

Mixing and Converting String Types (see page 2127)

3.2.4.10.48 Using TIniFile and TMemIniFile

The ini file format is still popular, many configuration files (such as the DSK Desktop settings file) are in this format. This format is especially useful in cross-platform applications, where you can't always count on a system Registry for storing configuration information. The VCL/RTL provides two classes, *TIniFile* and *TMemIniFile*, to make reading and writing ini files very easy.

TIniFile works directly with the ini file on disk while *TMemIniFile* buffers all changes in memory and does not write them to disk until you call the *UpdateFile* method.

When you instantiate the *TIniFile* or *TMemIniFile* object, you pass the name of the ini file as a parameter to the constructor. If the file does not exist, it is automatically created. You are then free to read values using the various read methods, such as *ReadString*, *ReadDate*, *ReadInteger*, or *ReadBool*. Alternatively, if you want to read an entire section of the ini file, you can use the *ReadSection* method. Similarly, you can write values using methods such as *WriteBool*, *WriteInteger*, *WriteDate*, or

WriteString.

Following is an example of reading configuration information from an ini file in a form's *OnCreate* event handler and writing values in the *OnClose* event handler.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create( ChangeFileExt( Application.ExeName, '.INI' ) );
  try
    Top     := Ini.ReadInteger( 'Form', 'Top', 100 );
 Left    := Ini.ReadInteger( 'Form', 'Left', 100 );
 Caption := Ini.ReadString( 'Form', 'Caption', 'New Form' );
    if Ini.ReadBool( 'Form', 'InitMax', false ) then
      WindowState = wsMaximized
    else
      WindowState = wsNormal;
  finally
    TIniFile.Free;
  end;
end;
procedure TForm1.FormClose(Sender: TObject; var Action TCloseAction)
var
  Ini: TIniFile;
begin
  Ini := TIniFile.Create( ChangeFileExt( Application.ExeName, '.INI' ) );
  try
    Ini.WriteInteger( 'Form', 'Top', Top);
    Ini.WriteInteger( 'Form', 'Left', Left);
 Ini.WriteString( 'Form', 'Caption', Caption );
    Ini.WriteBool( 'Form', 'InitMax', WindowState = wsMaximized );
  finally
    TIniFile.Free;
  end;
end;
__fastcall TForm1::TForm1(TComponent *Owner) : TForm(Owner)
{
TIniFile *ini;
ini = new TIniFile( ChangeFileExt( Application->ExeName, ".INI" ) );
Top     =  ini->ReadInteger( "Form", "Top", 100 );
Left    =  ini->ReadInteger( "Form", "Left", 100 );
Caption =  ini->ReadString( "Form", "Caption",
"Default Caption" );
ini->ReadBool( "Form", "InitMax", false ) ?
WindowState = wsMaximized :
WindowState = wsNormal;
delete ini;
}
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
TIniFile *ini;
ini = new TIniFile(ChangeFileExt( Application->ExeName, ".INI" ) );
   ini->WriteInteger( "Form", "Top", Top );
ini->WriteInteger( "Form", "Left", Left );
ini->WriteString ( "Form", "Caption", Caption );
ini->WriteBool   ( "Form", "InitMax",
WindowState == wsMaximized );
delete ini;
}
```

Each of the Read routines takes three parameters. The first parameter identifies the section of the ini file. The second parameter identifies the value you want to read, and the third is a default value in case the section or value doesn't exist in the ini file. Just as the Read methods gracefully handle the case when a section or value does not exist, the Write routines create the section and/or value if they do not exist. The example code creates an ini file the first time it is run that looks like this:

```
[Form]
Top=100
Left=100
Caption=Default Caption
InitMax=0
```

On subsequent execution of this application, the ini values are read in when the form is created and written back out in the *OnClose* event.

See Also

Using TRegistryIniFile (see page 2130)

3.2.4.10.49 Using File Streams

The *TFileStream* class enables applications to read from and write to a file on disk. Because *TFileStream* is a stream object, it shares the common stream methods. You can use these methods to read from or write to the file, copy data to or from other stream classes, and read or write components values. See Using streams (see page 2099) for details on the capabilities that files streams inherit by being stream classes.

In addition, file streams give you access to the file handle, so that you can use them with global file handling routines that require the file handle.

Creating and opening files using file streams

To create or open a file and get access to its handle, you simply instantiate a *TFileStream*. This opens or creates a specified file and provides methods to read from or write to it. If the file cannot be opened, the *TFileStream* constructor raises an exception.

```
constructor Create(const filename: string; Mode: Word);
__fastcall TFileStream(const AnsiString FileName, Word Mode);
```

The *Mode* parameter specifies how the file should be opened when creating the file stream. The *Mode* parameter consists of an open mode and a share mode OR'ed together. The open mode must be one of the following values:

Open modes

Value	Meaning
fmCreate	TFileStream a file with the given name. If a file with the given name exists, open the file in write mode.
fmOpenRead	Open the file for reading only.
fmOpenWrite	Open the file for writing only. Writing to the file completely replaces the current contents.
fmOpenReadWrite	Open the file to modify the current contents rather than replace them.

The share mode can be one of the following values with the restrictions listed below:

Share modes

Value	Meaning
fmShareCompat	Sharing is compatible with the way FCBs are opened (VCL applications only).
fmShareExclusive	Other applications can not open the file for any reason.
fmShareDenyWrite	Other applications can open the file for reading but not for writing.
fmShareDenyRead	Other applications can open the file for writing but not for reading (VCL applications only).
fmShareDenyNone	No attempt is made to prevent other applications from reading from or writing to the file.

Note that which share mode you can use depends on which open mode you used. The following table shows shared modes that are available for each open mode.

Shared modes available for each open mode

Open Mode	fmShareCompat (VCL)	fmShareExclusive	fmShareDenyWrite	fmShareDenyRead (VCL)	fmShareDenyNone
fmOpenRead	Can't use	Can't use	Available	Can't use	Available
fmOpenWrite	Available	Available	Can't use	Available	Available
fmOpenReadWrite	Available	Available	Available	Available	Available

The file open and share mode constants are defined in the *SysUtils* unit.

Using the file handle

When you instantiate *TFileStream* you get access to the file handle. The file handle is contained in the *Handle* property. On Windows, *Handle* is a Windows file handle. *Handle* is read-only and reflects the mode in which the file was opened. If you want to change the attributes of the file *Handle*, you must create a new file stream object.

Some file manipulation routines take a file handle as a parameter. Once you have a file stream, you can use the *Handle* property in any situation in which you would use a file handle. Be aware that, unlike handle streams, file streams close file handles when the object is destroyed.

See Also

Using Streams (see page 2099)

3.2.4.10.50 Accessing a Particular String

The *Strings* array property contains the strings in the list, referenced by a zero-based index. Because *Strings* is the default property for string lists, you can omit the *Strings* identifier when accessing the list; thus

```
StringList1.Strings[0] := 'This is the first string.';
StringList1->Strings[0] = "This is the first string.";
```

is equivalent to

```
StringList1[0] := 'This is the first string.';
(*StringList1)[0] = "This is the first string.";
```

See Also

Manipulating Strings in a List (see page 2139)

3.2.4.10.51 Adding a String to a List

To add a string to the end of a string list, call the *Add* method, passing the new string as the parameter. To insert a string into the list, call the *Insert* method, passing two parameters: the string and the index of the position where you want it placed. For example, to make the string "Three" the third string in a list, you would use:

```
Insert(2, 'Three');
StringList1->Insert(2, "Three");
```

To append the strings from one list onto another, call *AddStrings*:

```
StringList1.AddStrings(StringList2);  { append the strings from StringList2 to StringList1 }
StringList1->AddStrings(StringList2);  // append the strings from StringList2 to StringList1
```

See Also

Manipulating Strings in a List (see page 2139)

3.2.4.10.52 Associating Objects with a String List

In addition to the strings stored in its *Strings* property, a string list can maintain references to *objects*, which it stores in its *Objects* property. Like *Strings*, *Objects* is an array with a zero-based index. The most common use for *Objects* is to associate bitmaps with strings for owner-draw controls.

Use the *AddObject* or *InsertObject* method to add a string and an associated object to the list in a single step. *IndexOfObject* returns the index of the first string in the list associated with a specified object. Methods like *Delete*, *Clear*, and *Move* operate on both strings and objects; for example, deleting a string removes the corresponding object (if there is one).

To associate an object with an existing string, assign the object to the *Objects* property at the same index. You cannot add an object without adding a corresponding string.

See Also

Working with String Lists (see page 2140)

3.2.4.10.53 Copying a Complete String List

You can use the *Assign* method to copy strings from a source list to a destination list, overwriting the contents of the destination list. To append strings without overwriting the destination list, use *AddStrings*. For example,

```
Memo1.Lines.Assign(ComboBox1.Items);    { overwrites original strings }
Memo1->Lines->Assign(ComboBox1->Item)s;
```

copies the lines from a combo box into a memo (overwriting the memo), while

```
Memo1.Lines.AddStrings(ComboBox1.Items);   { appends strings to end }
Memo1->Lines->AddStrings(ComboBox1->Items);
```

appends the lines from the combo box to the memo.

When making local copies of a string list, use the *Assign* method. If you assign one string-list variable to another—

```
StringList1 := StringList2;
StringList1 = StringList2;
```

—the original string-list object will be lost, often with unpredictable results.

See Also

Manipulating Strings in a List (see page 2139)

3.2.4.10.54 Counting the Strings in a List

The read-only *Count* property returns the number of strings in the list. Since string lists use zero-based indexes, *Count* is one more than the index of the last string.

See Also

Manipulating Strings in a List (see page 2139)

3.2.4.10.55 Creating a New String List

A string list is typically part of a component. There are times, however, when it is convenient to create independent string lists, for example to store strings for a lookup table. The way you create and manage a string list depends on whether the list is short-term (constructed, used, and destroyed in a single routine) or long-term (available until the application shuts down). Whichever type of string list you create, remember that you are responsible for freeing the list when you finish with it.

Short-term string lists

If you use a string list only for the duration of a single routine, you can create it, use it, and destroy it all in one place. This is the safest way to work with string lists. Because the string-list object allocates memory for itself and its strings, you should use a **try...finally** block to ensure that the memory is freed even if an exception occurs.

To create a short-term string list:

1. Construct the string-list object.
2. In the try part of a **try...finally** block, use the string list.
3. In the finally part, free the string-list object.

The following event handler responds to a button click by constructing a string list, using it, and then destroying it.

```
procedure TForm1.Button1Click(Sender: TObject);
var  TempList: TStrings;{ declare the list }
begin
  TempList := TStringList.Create;{ construct the list object }
  try    { use the string list }
  finally    TempList.Free;{ destroy the list object }
  end;
end;
void __fastcall TForm1::ButtonClick1(TObject *Sender)
{
   TStringList *TempList = new TStringList; // declare the list
   try{
      //use the string list
   }
   __finally{
delete TempList; // destroy the list object
   }
}
```

Long-term string lists

If a string list must be available at any time while your application runs, construct the list at start-up and destroy it before the application terminates.

To create a long-term string list:

1. In the unit file for your application's main form, add a field of type *TStrings* to the form's declaration.
2. Write an event handler for the main form's OnCreate event that executes before the form appears. It should create a string list and assign it to the field you declared in the first step.
3. Write an event handler that frees the string list for the form's *OnClose* event.

This example uses a long-term string list to record the user's mouse clicks on the main form, then saves the list to a file before the application terminates.

```
unit Unit1;
interface
uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;
type  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y:
Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    ClickList: TStrings;{ declare the field }
  end;
var  Form1: TForm1;
```

```
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
        ClickList := TStringList.Create;{ construct the list }
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    ClickList.SaveToFile(ChangeFileExt(Application.ExeName, '.log'));{ save the list }
 ClickList.Free;{ destroy the list object }
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  Shift: TShiftState; X,
Y: Integer);
begin
    ClickList.Add(Format('Click at (%d, %d)', [X, Y]));{ add a string to the list }
end;

end.
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
: TForm(Owner)
{
ClickList = new TStringList;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender, TCloseAction &Action)
{
ClickList->SaveToFile(ChangeFileExt(Application->ExeName, ".log"));//Save the list
delete ClickList;
}
//---------------------------------------------------------------------------
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
TShiftState Shift, int X, int Y)
{
TVarRec v[] = {X,Y};
ClickList->Add(Format("Click at (%d, %d)",v,ARRAYSIZE(v) - 1));//add a string to the list
```

Note: Although you can use events such as OnCreate and OnDestroy to allocate and free classes, using the constructor and destructor for a class is generally safer coding practice.

See Also

Manipulating Strings in a List (see page 2139)

Loading and Saving String Lists (see page 2138)

Associating Objects with a String List (see page 2135)

3.2.4.10.56 Deleting a String from a List

To delete a string from a string list, call the list's *Delete* method, passing the index of the string you want to delete. If you don't know the index of the string you want to delete, use the *IndexOf* method to locate it. To delete all the strings in a string list, use the *Clear* method.

The following example uses *IndexOf* and *Delete* to find and delete a string:

```
with ListBox1.Items do
  begin
    FoundIndex := IndexOf('bureaucracy');
    if FoundIndex > -1 then
      Delete(FoundIndex);
  end;
int BIndex = ListBox1->Items->IndexOf("bureaucracy");
if (BIndex > -1)
  ListBox1->Items->Delete(BIndex);
```

See Also

Manipulating Strings in a List (see page 2139)

3.2.4.10.57 Locating Items in a String List

To locate a string in a string list, use the *IndexOf* method. *IndexOf* returns the index of the first string in the list that matches the parameter passed to it, and returns –1 if the parameter string is not found. *IndexOf* finds exact matches only; if you want to match partial strings, you must iterate through the string list yourself.

For example, you could use *IndexOf* to determine whether a given file name is found among the Items of a list box:

```
if FileListBox1.Items.IndexOf('TargetFileName') > -1 ...
if (FileListBox1->Items->IndexOf("TargetFileName") > -1) ...
```

See Also

Manipulating Strings in a List (see page 2139)

3.2.4.10.58 Iterating Through Strings in a List

To iterate through the strings in a list, use a **for** loop that runs from zero to *Count* –1.

The following example converts each string in a list box to uppercase characters.

```
procedure TForm1.Button1Click(Sender: TObject);var  Index: Integer;
begin
  for Index := 0 to ListBox1.Items.Count - 1 do     ListBox1.Items[Index] :=
UpperCase(ListBox1.Items[Index]);
end;
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  for (int i = 0; i < ListBox1->Items->Count; i++)
    ListBox1->Items->Strings[i] = UpperCase(ListBox1->Items->Strings[i]);
}
```

See Also

Manipulating Strings in a List (see page 2139)

3.2.4.10.59 Loading and Saving String Lists

String-list objects provide *SaveToFile* and *LoadFromFile* methods that let you store a string list in a text file and load a text file into a string list. Each line in the text file corresponds to a string in the list. Using these methods, you could, for example, create a simple text editor by loading a file into a memo component, or save lists of items for combo boxes.

The following example loads a copy of the MyFile.ini file into a memo field and makes a backup copy called MyFile.bak.

```
procedure EditWinIni;
var  FileName: string;{ storage for file name }
```

```
begin  FileName := 'c:\Program Files\MyProgram\MyFile.ini'{ set the file name }


  with Form1.Memo1.Lines do   begin
    LoadFromFile(FileName);{ load from file }
    SaveToFile(ChangeFileExt(FileName, '.bak'));{ save into backup file }
  end;
end;
void __fastcall EditWinIni()
{
  AnsiString FileName = "C:\\Program Files\\MyFile.ini";
  Form1->Memo1->Lines->LoadFromFile(FileName);      // load from file
```

See Also

Manipulating Strings in a List (see page 2139)

Creating a New String List (see page 2135)

Associating Objects with a String List (see page 2135)

3.2.4.10.60 Manipulating Strings in a List

Operations commonly performed on string lists include:

- Counting the strings in a list (see page 2135)
- Accessing a particular string (see page 2134)
- Finding the position of a string in the list (see page 2138)
- Iterating through strings in a list (see page 2138)
- Adding a string to a list (see page 2134)
- Moving a string within a list
- Deleting a string from a list (see page 2137)
- Copying a complete string list (see page 2135)

See Also

Loading and Saving String Lists (see page 2138)

Creating a New String List (see page 2135)

Working with String Lists (see page 2140)

Associating Objects with a String List (see page 2135)

3.2.4.10.61 Working with Lists

The VCL/RTL includes many classes that represents lists or collections of items. They vary depending on the types of items they contain, what operations they support, and whether they are persistent.

The following table lists various list classes, and indicates the types of items they contain:

Object	Maintains
TList	A list of pointers
TThreadList	A thread-safe list of pointers
TBucketList	A hashed list of pointers

TObjectBucketList	A hashed list of object instances
TObjectList	A memory-managed list of object instances
TComponentList	A memory-managed list of components (that is, instances of classes descended from *TComponent*)
TClassList	A list of class references
TInterfaceList	A list of interface pointers.
TQueue	A first-in first-out list of pointers
TStack	A last-in first-out list of pointers
TObjectQueue	A first-in first-out list of objects
TObjectStack	A last-in first-out list of objects
TCollection	Base class for many specialized classes of typed items.
TStringList	A list of strings
THashedStringList	A list of strings with the form Name=Value, hashed for performance.

See Also

Working with String Lists (see page 2140)

Common List Operations (see page 2091)

Persistent Lists (see page 2105)

Working with Strings (see page 2142)

3.2.4.10.62 Working with String Lists

One of the most commonly used types of list is a list of character strings. Examples include items in a combo box, lines in a memo, names of fonts, and names of rows and columns in a string grid. The VCL/RTL provides a common interface to any list of strings through an object called *TStrings* and its descendants such as *TStringList* and *THashedStringList*. *TStringList* implements the abstract properties and methods introduced by *TStrings*, and introduces properties, events, and methods to

- Sort the strings in the list.
- Prohibit duplicate strings in sorted lists.
- Respond to changes in the contents of the list.

In addition to providing functionality for maintaining string lists, these objects allow easy interoperability; for example, you can edit the lines of a memo (which are a *TStrings* descendant) and then use these lines as items in a combo box (also a *TStrings* descendant).

A string-list property appears in the **Object Inspector** with *TStrings* in the Value column. Double-click *TStrings* to open the String List editor, where you can edit, add, or delete lines.

You can also work with string-list objects at runtime to perform such tasks as

- Loading and saving string lists (see page 2138)
- Creating a new string list (see page 2135)
- Manipulating strings in a list (see page 2139)
- Associating objects with a string list (see page 2135)

See Also

Working with Lists (see page 2139)

Loading and Saving String Lists (see page 2138)

3.2.4.10.63 Wide Character Routines

Wide strings contain two bytes per element and are used in a variety of situations. You may also choose to use wide strings because they simplify some of the string-handling issues in applications that have multiple target locales. Using a wide character encoding scheme has the advantage that you can make many of the usual assumptions about strings that do not work for MBCS systems, which are single byte strings. For wide strings, there is a direct relationship between the number of bytes in the string and the number of elements in the string. In a MBCS string, you have to be concerned about cutting characters in half or mistaking the second part of a character for the start of a different character. There is a similar issue for a wide string. Although elements are all two bytes, characters not in the Basic Multilingual Plane (BMP) require two elements.

Two types represent wide strings: `UnicodeString` and `WideString`. The `WideString` format is essentially the same as a Windows `BSTR`. `WideString` is appropriate for use in COM applications. `WideString` is not reference counted, so `UnicodeString` is more flexible and efficient in other types of applications. In addition, more utility functions are available for `UnicodeString` than `WideString`, so `UnicodeString` is generally preferred. This topic deals with `WideString`, not `UnicodeString`.

VCL now uses the `UnicodeString` type; it no longer represents string values as single byte or MBCS strings.

The following functions convert between standard single-byte character strings (or MBCS strings) and Unicode strings:

- StringToWideChar
- WideCharLenToString
- WideCharLenToStrVar
- WideCharToString
- WideCharToStrVar

In addition, the following functions translate between WideStrings and other representations:

- UCS4StringToWideString
- WideStringToUCS4String
- VarToWideStr
- VarToWideStrDef

The following routines work directly with WideStrings:

- WideCompareStr
- WideCompareText
- WideSameStr
- WideSameText
- WideFmtStr
- WideFormat
- WideLowerCase
- WideUpperCase

Finally, some routines include overloads for working with wide strings:

- UniqueString
- Length
- Trim
- TrimLeft

- TrimRight

See Also

Creating International Applications (see page 1923)

Enabling Unicode in Your Applications

Unicode in RAD Studio

3.2.4.10.64 Working with Files

The VCL/RTL supports several ways of working with files. In addition to using file streams, there are several runtime library routines for performing file I/O. Both file streams and the global routines for reading from and writing to files are described in Approaches to file I/O (see page 2091).

In addition to input/output operations, you may want to manipulate files on disk. Support for operations on the files themselves rather than their contents is described in Manipulating files (see page 2127).

Note: When writing cross-platform applications, remember that although the Delphi language is not case sensitive, the Linux operating system is. When using objects and routines that work with files, be attentive to the case of file names.

See Also

Using Streams (see page 2099)

Working with ini Files and the System Registry (see page 2099)

3.2.4.10.65 Working with Strings

The runtime library provides many specialized string-handling routines specific to a string type. These are routines for strings (`AnsiString` and `UnicodeString`), wide strings (`WideString`), and null-terminated strings (`PChars`). Routines that deal with null-terminated strings use the null-termination to determine the length of the string. There are no categories of routines listed for *ShortString* types. However, some built-in compiler routines deal with the *ShortString* type. These include, for example, the `Low` and `High` standard functions. For more details about the various string types, see the *Delphi Language Guide*.

The following topics provide an overview of many of the string-handling routines in the runtime library:

- Commonly used long string routines (see page 2118)
- Commonly used routines for null-terminated strings (see page 2121)
- Wide character routines (see page 2141)

See Also

Enabling Unicode in Your Applications

Unicode in RAD Studio

Working with String Lists (see page 2140)

3.2.4.11 Working with components

Topics

Name	Description
Adding Custom Components to the Tool Palette (see page 2144)	You can install custom components—written by yourself or third parties—on the **Tool palette** and use them in your applications. To write a custom component, see Overview of component creation (see page 1313). To install an existing component, see Installing component packages (see page 2217).

Associating Menu Events with Event Handlers (see page 2144)	The Menu Designer, along with the *MainMenu* and *PopupMenu* components, make it easy to supply your application with drop-down and pop-up menus. For the menus to work, however, each menu item must respond to the *OnClick* event, which occurs whenever the user chooses the menu item or presses its accelerator or shortcut key. This topic explains how to associate event handlers with menu items. For information about the Menu Designer and related components, see Creating and managing menus (see page 1975).
Calling Methods (see page 2145)	Methods are called just like ordinary procedures and functions. For example, visual controls have a *Repaint* method that refreshes the control's image on the screen. You could call the *Repaint* method in a draw-grid object like this:
Deleting Event Handlers (see page 2145)	When you delete a component from a form using the Form Designer, the Code editor removes the component from the form's type declaration. It does not, however, delete any associated methods from the unit file, since these methods may still be called by other components on the form. You can manually delete a method—such as an event handler—but if you do so, be sure to delete both the method's forward declaration (in the unit's **interface** section) and its implementation (in the **implementation** section). Otherwise you'll get a compiler error when you build your project.
Displaying and Coding Shared Events (see page 2145)	When components share events, you can display their shared events in the **Object Inspector**. First, select the components by holding down the `Shift` key and clicking on them in the Form Designer; then choose the Events tab in the **Object Inspector**. From the Value column in the **Object Inspector**, you can now create a new event handler for, or assign an existing event handler to, any of the shared events.
Associating an Event with an Existing Event Handler (see page 2146)	You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop-down menu commands. When a button initiates the same action as a menu command, you can write a single event handler and assign it to both the button's and the menu item's *OnClick* event.
Generating a New Event Handler (see page 2146)	You can generate skeleton event handlers for forms and other components.
Generating a Handler for a Component's Default Event (see page 2147)	Some components have a *default* event, which is the event the component most commonly needs to handle. For example, a button's default event is *OnClick*. To create a default event handler, double-click the component in the Form Designer; this generates a skeleton event-handling procedure and opens the Code editor with the cursor in the body of the procedure, where you can easily add code. Not all components have a default event. Some components, such as *TBevel*, don't respond to any events. Other components respond differently when you double-click them in the Form Designer. For example, many components open... more (see page 2147)
Locating Event Handlers (see page 2147)	If you generated a default event handler for a component by double-clicking it in the Form Designer, you can locate that event handler in the same way. Double-click the component, and the Code editor opens with the cursor at the beginning of the event-handler body.
Setting Properties at Design Time (see page 2147)	When you select a component on a form at design time, the **Object Inspector** displays its published properties and (when appropriate) allows you to edit them. Use the `Tab` key to toggle between the left-hand Property column and the right-hand Value column. When the cursor is in the Property column, you can navigate to any property by typing the first letters of its name. For properties of Boolean or enumerated types, you can choose values from a drop-down list or toggle their settings by double-clicking in Value column. If a plus (+) symbol appears next to a property name, clicking... more (see page 2147)
Setting Component Properties (see page 2148)	You can set component properties at design time or at runtime: • To set published properties at design time, you can use the **Object Inspector** and, in some cases, special property editors. • To set properties at runtime, assign their values in your application source code For information about the properties of each component, see the VCL library in the online Help.
Setting Properties at Runtime (see page 2148)	Any writable property can be set at runtime in your source code. For example, you can dynamically assign a caption to a form:

Using Property Editors (see page 2148)	Some properties, such as *Font*, have special property editors. Such properties appear with ellipsis marks (...) next to their values when the property is selected in the **Object Inspector**. To open the property editor, double-click in the Value column, click the ellipsis mark, or type `Ctrl+Enter` when focus is on the property or its value. With some components, double-clicking the component on the form also opens a property editor. Property editors let you set complex properties from a single dialog box. They provide input validation and often let you preview the results of an assignment.
Using the Sender Parameter (see page 2148)	In an event handler, the *Sender* parameter indicates which component received the event and therefore called the handler. Sometimes it is useful to have several components share an event handler that behaves differently depending on which component calls it. You can do this by using the *Sender* parameter in an **if...then...else** statement. For example, the following code displays the title of the application in the caption of a dialog box only if the *OnClick* event was received by *Button1*.
Working with Events and Event Handlers (see page 2149)	Almost all the code you write is executed, directly or indirectly, in response to events. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event—called an *event handler*—is a Delphi procedure. The sections that follow show how to: • Generate a new event handler. (see page 2146) • Generate a handler for a component's default event. (see page 2147) • Locate event handlers. (see page 2147) • Associate an event with an existing event handler. (see page 2146) • Associate menu events with event handlers. (see page 2144) • Delete event handlers. (see page 2145)

3.2.4.11.1 Adding Custom Components to the Tool Palette

You can install custom components—written by yourself or third parties—on the **Tool palette** and use them in your applications. To write a custom component, see Overview of component creation (see page 1313). To install an existing component, see Installing component packages (see page 2217).

See Also

Installing Component Packages (see page 2217)

Overview of Component Creation (see page 1313)

3.2.4.11.2 Associating Menu Events with Event Handlers

The Menu Designer, along with the *MainMenu* and *PopupMenu* components, make it easy to supply your application with drop-down and pop-up menus. For the menus to work, however, each menu item must respond to the *OnClick* event, which occurs whenever the user chooses the menu item or presses its accelerator or shortcut key. This topic explains how to associate event handlers with menu items. For information about the Menu Designer and related components, see Creating and managing menus (see page 1975).

To create an event handler for a menu item:

1. Open the Menu Designer by double-clicking on a MainMenu or PopupMenu component.
2. Select a menu item in the Menu Designer. In the **Object Inspector**, make sure that a value is assigned to the item's Name property.
3. From the Menu Designer, double-click the menu item. The Code editor opens with the cursor inside the skeleton event

handler, or the **begin...end** block.

4. At the cursor, type the code that you want to execute when the user selects the menu command.

To associate a menu item with an existing OnClick event handler:

1. Open the Menu Designer by double-clicking a MainMenu or PopupMenu component.
2. Select a menu item in the Menu Designer. In the **Object Inspector**, make sure that a value is assigned to the item's Name property.
3. On the Events page of the **Object Inspector**, click the down arrow in the Value column next to OnClick to open a list of previously written event handlers. (The list includes only event handlers written for OnClick events on this form.) Select from the list by clicking an event handler name.

See Also

Working with Events and Event Handlers (see page 2149)

Deleting Event Handlers (see page 2145)

3.2.4.11.3 Calling Methods

Methods are called just like ordinary procedures and functions. For example, visual controls have a *Repaint* method that refreshes the control's image on the screen. You could call the *Repaint* method in a draw-grid object like this:

```
DrawGrid1.Repaint;
DrawGrid1->Repaint;
```

As with properties, the scope of a method name determines the need for qualifiers. If you want, for example, to repaint a form within an event handler of one of the form's child controls, you don't have to prepend the name of the form to the method call:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Repaint;
end;
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Repaint;
}
```

For more information about scope, see Scope and Qualifiers (see page 2070).

3.2.4.11.4 Deleting Event Handlers

When you delete a component from a form using the Form Designer, the Code editor removes the component from the form's type declaration. It does not, however, delete any associated methods from the unit file, since these methods may still be called by other components on the form. You can manually delete a method—such as an event handler—but if you do so, be sure to delete both the method's forward declaration (in the unit's **interface** section) and its implementation (in the **implementation** section). Otherwise you'll get a compiler error when you build your project.

See Also

Working with Events and Event Handlers (see page 2149)

3.2.4.11.5 Displaying and Coding Shared Events

When components share events, you can display their shared events in the **Object Inspector**. First, select the components by holding down the `Shift` key and clicking on them in the Form Designer; then choose the Events tab in the **Object Inspector**. From the Value column in the **Object Inspector**, you can now create a new event handler for, or assign an existing event

handler to, any of the shared events.

See Also

Working with Events and Event Handlers (see page 2149)

Associating an Event with an Existing Event Handler (see page 2146)

Using Action Lists (see page 2007)

3.2.4.11.6 Associating an Event with an Existing Event Handler

You can reuse code by writing event handlers that respond to more than one event. For example, many applications provide speed buttons that are equivalent to drop-down menu commands. When a button initiates the same action as a menu command, you can write a single event handler and assign it to both the button's and the menu item's *OnClick* event.

To associate an event with an existing event handler

1. On the form, select the component whose event you want to handle.
2. On the Events page of the **Object Inspector**, select the event to which you want to attach a handler.
3. Click the down arrow in the Value column next to the event to open a list of previously written event handlers. (The list includes only event handlers written for events of the same name on the same form.) Select from the list by clicking an event-handler name.

The previous procedure is an easy way to reuse event handlers. Action lists (see page 2007) and in the VCL, action bands (see page 2002), however, provide powerful tools for centrally organizing the code that responds to user commands. Action lists can be used in cross-platform applications, whereas action bands cannot.

See Also

Working with Events and Event Handlers (see page 2149)

Using the Sender Parameter (see page 2148)

Displaying and Coding Shared Events (see page 2145)

Associating Menu Events with Event Handlers (see page 2144)

3.2.4.11.7 Generating a New Event Handler

You can generate skeleton event handlers for forms and other components.

To create an event handler:

1. Select a component.
2. Click the Events tab in the **Object Inspector**. The Events page of the **Object Inspector** displays all events defined for the component.
3. Select the event you want, then double-click the Value column or press `Ctrl+Enter`. The Code editor opens with the cursor inside the skeleton event handler, or **begin...end** block.
4. At the cursor, type the code that you want to execute when the event occurs.

See Also

Working with Events and Event Handlers (see page 2149)

Generating a Handler for a Component's Default Event (see page 2147)

3.2.4.11.8 Generating a Handler for a Component's Default Event

Some components have a *default* event, which is the event the component most commonly needs to handle. For example, a button's default event is *OnClick*. To create a default event handler, double-click the component in the Form Designer; this generates a skeleton event-handling procedure and opens the Code editor with the cursor in the body of the procedure, where you can easily add code.

Not all components have a default event. Some components, such as *TBevel*, don't respond to any events. Other components respond differently when you double-click them in the Form Designer. For example, many components open a default property editor or other dialog when they are double-clicked at design time.

See Also

Working with Events and Event Handlers (see page 2149)

Generating a New Event Handler (see page 2146)

Locating Event Handlers (see page 2147)

Associating an Event with an Existing Event Handler (see page 2146)

3.2.4.11.9 Locating Event Handlers

If you generated a default event handler for a component by double-clicking it in the Form Designer, you can locate that event handler in the same way. Double-click the component, and the Code editor opens with the cursor at the beginning of the event-handler body.

To locate an event handler that's not the default:

1. In the form, select the component whose event handler you want to locate.
2. In the **Object Inspector**, click the Events tab.
3. Select the event whose handler you want to view and double-click in the Value column. The Code editor opens with the cursor inside the skeleton event-handler.

See Also

Working with Events and Event Handlers (see page 2149)

Generating a New Event Handler (see page 2146)

3.2.4.11.10 Setting Properties at Design Time

When you select a component on a form at design time, the **Object Inspector** displays its published properties and (when appropriate) allows you to edit them. Use the Tab key to toggle between the left-hand Property column and the right-hand Value column. When the cursor is in the Property column, you can navigate to any property by typing the first letters of its name. For properties of Boolean or enumerated types, you can choose values from a drop-down list or toggle their settings by double-clicking in Value column.

If a plus (+) symbol appears next to a property name, clicking the plus symbol or typing '+' when the property has focus displays a list of subvalues for the property. Similarly, if a minus (-) symbol appears next to the property name, clicking the minus symbol or typing '-' hides the subvalues.

By default, properties in the Legacy category are not shown; to change the display filters, right-click in the **Object Inspector** and choose View.

When more than one component is selected, the **Object Inspector** displays all properties—except *Name*—that are shared by

the selected components. If the value for a shared property differs among the selected components, the **Object Inspector** displays either the default value or the value from the first component selected. When you change a shared property, the change applies to all selected components.

Changing code-related properties, such as the name of an event handler, in the **Object Inspector** automatically changes the corresponding source code. In addition, changes to the source code, such as renaming an event handler method in a form class declaration, is immediately reflected in the **Object Inspector**.

See Also

Using Property Editors (see page 2148)

3.2.4.11.11 Setting Component Properties

You can set component properties at design time or at runtime:

- To set published properties at design time, you can use the **Object Inspector** and, in some cases, special property editors.
- To set properties at runtime, assign their values in your application source code

For information about the properties of each component, see the VCL library in the online Help.

See Also

Setting Properties at Design Time (see page 2147)

Setting Properties at Runtime (see page 2148)

3.2.4.11.12 Setting Properties at Runtime

Any writable property can be set at runtime in your source code. For example, you can dynamically assign a caption to a form:

```
Form1.Caption := MyString;
Form1->Caption = MyString;
```

See Also

Setting Component Properties (see page 2148)

3.2.4.11.13 Using Property Editors

Some properties, such as *Font*, have special property editors. Such properties appear with ellipsis marks (...) next to their values when the property is selected in the **Object Inspector**. To open the property editor, double-click in the Value column, click the ellipsis mark, or type `Ctrl+Enter` when focus is on the property or its value. With some components, double-clicking the component on the form also opens a property editor.

Property editors let you set complex properties from a single dialog box. They provide input validation and often let you preview the results of an assignment.

3.2.4.11.14 Using the Sender Parameter

In an event handler, the *Sender* parameter indicates which component received the event and therefore called the handler. Sometimes it is useful to have several components share an event handler that behaves differently depending on which component calls it. You can do this by using the *Sender* parameter in an **if...then...else** statement. For example, the following code displays the title of the application in the caption of a dialog box only if the *OnClick* event was received by *Button1*.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
```

```
if Sender = Button1 then
  AboutBox.Caption := 'About ' + Application.Title
else
  AboutBox.Caption := '';
AboutBox.ShowModal;
end;
```

See Also

Working with Events and Event Handlers (see page 2149)

Associating an Event with an Existing Event Handler (see page 2146)

3.2.4.11.15 Working with Events and Event Handlers

Almost all the code you write is executed, directly or indirectly, in response to events. An event is a special kind of property that represents a runtime occurrence, often a user action. The code that responds directly to an event—called an *event handler*—is a Delphi procedure. The sections that follow show how to:

- Generate a new event handler. (see page 2146)
- Generate a handler for a component's default event. (see page 2147)
- Locate event handlers. (see page 2147)
- Associate an event with an existing event handler. (see page 2146)
- Associate menu events with event handlers. (see page 2144)
- Delete event handlers. (see page 2145)

See Also

Associating Menu Events with Event Handlers (see page 2144)

Locating Event Handlers (see page 2147)

Calling Methods (see page 2145)

Setting Component Properties (see page 2148)

3.2.4.12 Working with controls

Topics

Name	Description
Accepting Dragged Items (see page 2153)	When the user drags something over a control, that control receives an *OnDragOver* event, at which time it must indicate whether it can accept the item if the user drops it there. The drag cursor changes to indicate whether the control can accept the dragged item. To accept items dragged over a control, attach an event handler to the control's *OnDragOver* event. The drag-over event has a parameter called *Accept* that the event handler can set to *True* if it will accept the item. *Accept* changes the cursor type to an accept cursor or not. The drag-over event has other... more (see page 2153)
Adding Graphical Objects to a String List (see page 2154)	Every string list has the ability to hold a list of objects in addition to its list of strings. You can also add graphical objects of varying sizes to a string list. For example, in a file manager application, you may want to add bitmaps indicating the type of drive along with the letter of the drive. To do that, you need to add the bitmap images to the application, then copy those images into the proper places in the string list as described in the following sections. Note that you can also organize graphical objects using an image list... more (see page 2154)

Adding Graphics to Controls (see page 2154)	Several controls let you customize the way the control is rendered. These include list boxes, combo boxes, menus, headers, tab controls, list views, status bars, tree views, and toolbars. Instead of using the standard method of drawing a control or its items, the control's owner (generally, the form) draws them at runtime. The most common use for owner-draw controls is to provide graphics instead of, or in addition to, text for items. For information on using owner-draw to add images to menus, see Adding images to menu items. (see page 1967). All owner-draw controls contain lists of items. Usually, those lists are... more (see page 2154)
Adding Images to a String List (see page 2154)	Once you have graphical images in an application, you can associate them with the strings in a string list. You can either add the objects at the same time as the strings, or associate objects with existing strings. The preferred method is to add objects and strings at the same time, if all the needed data is available. The following example shows how you might want to add images to a string list. This is part of a file manager application where, along with a letter for each valid drive, it adds a bitmap indicating each drive's type. The *OnCreate... more (see page 2154)*
Adding Images to an Application (see page 2155)	An image control is a nonvisual control that contains a graphical image, such as a bitmap. You use image controls to display graphical images on a form. You can also use them to hold hidden images that you'll use in your application. For example,
Adding Scroll Bars at Runtime (see page 2156)	Rich edit and memo components can contain horizontal or vertical scroll bars, or both, as needed. When word wrapping is enabled, the component needs only a vertical scroll bar. If the user turns off word wrapping, the component might also need a horizontal scroll bar, since text is not limited by the right side of the editor.
Adding the Clipboard Object (see page 2156)	Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. *TClipboard* object encapsulates a clipboard (such as the Windows Clipboard) and includes methods for cutting, copying, and pasting text (and other formats, including graphics). The *Clipboard* object is declared in the *Clipbrd* unit.
Changing the Drag Mouse Pointer (see page 2157)	You can customize the appearance of the mouse pointer during drag operations by setting the source component's *DragCursor* property (VCL only).
Controlling How Child Controls Are Docked (see page 2157)	A docking site automatically accepts child controls when they are released over the docking site. For most controls, the first child is docked to fill the client area, the second splits that into separate regions, and so on. Page controls dock children into new tab sheets (or merge in the tab sheets if the child is another page control). Three events allow docking sites to further constrain how child controls are docked:
Controlling How Child Controls Are Undocked (see page 2158)	A docking site automatically allows child controls to be undocked when they are dragged and have a *DragMode* property of *dmAutomatic*. Docking sites can respond when child controls are dragged off, and even prevent the undocking, in an *OnUnDock* event handler:
Controlling How Child Controls Respond to Drag-and-dock Operations (see page 2158)	Dockable child controls have two events that occur during drag-and-dock operations: *OnStartDock*, analogous to the *OnStartDrag* event of a drag-and-drop operation, allows the dockable child control to create a custom drag object. *OnEndDock*, like *OnEndDrag*, occurs when the dragging terminates.
Customizing Drag and Drop with a Drag Object (see page 2158)	You can use a *TDragObject* descendant to customize an object's drag-and-drop behavior. The standard drag-over and drag-and-drop events indicate the source of the dragged item and the coordinates of the mouse cursor over the accepting control. To get additional state information, derive a custom drag object from *TDragObject* or *TDragObjectEx* (VCL only) and override its virtual methods. Create the custom drag object in the *OnStartDrag* event. Normally, the source parameter of the drag-over and drag-and-drop events is the control that starts the drag operation. If different kinds of control can start an operation involving the same kind of data, the... more (see page 2158)
Cutting, Copying, and Pasting Text (see page 2159)	Applications that use the *Clipbrd* unit can cut, copy, and paste text, graphics, and objects through the clipboard. The edit components that encapsulate the standard text-handling controls all have methods built into them for interacting with the clipboard. To cut, copy, or paste text with the clipboard, call the edit component's *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods, respectively. For example, the following code attaches event handlers to the *OnClick* events of the **Edit ▶ Cut**, **Edit ▶ Copy**, and **Edit ▶ Paste** commands, respectively:
Deleting Selected Text (see page 2160)	You can delete the selected text in an edit component without cutting it to the clipboard. To do so, call the *ClearSelection* method. For example, if you have a Delete item on the Edit menu, your code could look like this:

Disabling Menu Items (see page 2160)	It is often useful to disable menu commands without removing them from the menu. For example, in a text editor, if there is no text currently selected, the Cut, Copy, and Delete commands are inapplicable. An appropriate time to enable or disable menu items is when the user selects the menu. To disable a menu item, set its *Enabled* property to *False*. In the following example, an event handler is attached to the *OnClick* event for the Edit item on a child form's menu bar. It sets *Enabled* for the Cut, Copy, and Delete menu items on the Edit... more (see page 2160)
Implementing Drag and Drop in Controls (see page 2160)	Drag-and-drop is often a convenient way for users to manipulate objects. You can let users drag an entire control, or let them drag items from one control—such as a list box or tree view—into another. • Starting a drag operation (see page 2168) • Accepting dragged items (see page 2153) • Dropping items (see page 2162) • Ending a drag operation (see page 2162) • Customizing drag and drop with a drag object (see page 2158) • Changing the drag mouse pointer (see page 2157)
Drawing Owner-draw Items (see page 2161)	When an application needs to draw or redraw an owner-draw control, the operating system generates draw-item events for each visible item in the control. Depending on the control, the item may also receive draw events for the item as a part of the item. To draw each item in an owner-draw control, attach an event handler to the draw-item event for that control. The names of events for owner drawing typically start with one of the following: • *OnDraw*, such as *OnDrawItem* or *OnDrawCell* • *OnCustomDraw*, such as *OnCustomDrawItem* • *OnAdvancedCustomDraw*, such as *OnAdvancedCustomDrawItem* The draw-item event contains parameters identifying... more (see page 2161)
Drawing Owner-drawn Items (see page 2161)	When you indicate that a control is owner-drawn, either by setting a property or supplying a custom draw event handler, the control is no longer drawn on the screen. Instead, the operating system generates events for each visible item in the control. Your application handles the events to draw the items.
Dropping Items (see page 2162)	If a control indicates that it can accept a dragged item, it needs to handle the item should it be dropped. To handle dropped items, attach an event handler to the *OnDragDrop* event of the control accepting the drop. Like the drag-over event, the drag-and-drop event indicates the source of the dragged item and the coordinates of the mouse cursor over the accepting control. The latter parameter allows you to monitor the path an item takes while being dragged; you might, for example, want to use this information to change the color of components if an item is dropped. In... more (see page 2162)
Ending a Drag Operation (see page 2162)	A drag operation ends when the item is either successfully dropped or released over a control that cannot accept it. At this point an end-drag event is sent to the control from which the drag was initiated. To enable a control to respond when items have been dragged from it, attach an event handler to the control's *OnEndDrag* event. The most important parameter in an *OnEndDrag* event is called *Target*, which indicates which control, if any, accepts the drop. If *Target* is **nil**, it means no control accepts the dragged item. The *OnEndDrag* event also includes the coordinates... more (see page 2162)
Handling the OnPopup Event (see page 2163)	You may want to adjust pop-up menu items before displaying the menu, just as you may want to enable or disable items on a regular menu. With a regular menu, you can handle the *OnClick* event for the item at the top of the menu. With a pop-up menu, however, there is no top-level menu bar, so to prepare the pop-up menu commands, you handle the event in the menu component itself. The pop-up menu component provides an event just for this purpose, called *OnPopup*.

Implementing Drag and Dock in Controls (see page 2163)	Descendants of *TWinControl* can act as docking sites and descendants of *TControl* can act as child windows that are docked into docking sites. For example, to provide a docking site at the left edge of a form window, align a panel to the left edge of the form and make the panel a docking site. When dockable controls are dragged to the panel and released, they become child controls of the panel. • Making a windowed control a docking site (see page 2164) • Making a control a dockable child (see page 2164) • Controlling how child controls are docked (see page 2157) • Controlling how child controls are undocked (see page 2158) • Controlling how child (see page 2158)... more (see page 2163)
Making a Control a Dockable Child (see page 2164)	
Making a Windowed Control a Docking Site (see page 2164)	
Providing a Pop-up Menu (see page 2164)	Pop-up, or local, menus are a common ease-of-use feature for any application. They enable users to minimize mouse movement by clicking the right mouse button in the application workspace to access a list of frequently used commands. In a text editor application, for example, you can add a pop-up menu that repeats the Cut, Copy, and Paste editing commands. These pop-up menu items can use the same event handlers as the corresponding items on the Edit menu. You don't need to create accelerator or shortcut keys for pop-up menus because the corresponding regular menu items generally already have shortcuts. A... more (see page 2164)
Selecting All Text (see page 2165)	The *SelectAll* method selects the entire contents of an edit control, such as a rich edit or memo component. This is especially useful when the component's contents exceed the visible area of the component. In most other cases, users select text with either keystrokes or mouse dragging. To select the entire contents of a rich edit or memo control, call the *RichEdit1* control's *SelectAll* method. For example:
Selecting Text (see page 2165)	For text in an edit control, before you can send any text to the clipboard, that text must be selected. Highlighting of selected text is built into the edit components. When the user selects text, it appears highlighted. The table below lists properties commonly used to handle selected text. ***Properties of selected text***
Indicating That a Control Is Owner-drawn (see page 2166)	To customize the drawing of a control, you must supply event handlers that render the control's image when it needs to be painted. Some controls receive these events automatically. For example, list views, tree views, and toolbars all receive events at various stages in the drawing process without your having to set any properties. These events have names such as *OnCustomDraw* or *OnAdvancedCustomDraw*. Other controls, however, require you to set a property before they receive owner-draw events. List boxes, combo boxes, header controls, and status bars have a property called *Style*. *Style* determines whether the control uses the... more (see page 2166)
Setting Text Alignment (see page 2166)	In a rich edit or memo component, text can be left- or right-aligned or centered. To change text alignment, set the edit component's *Alignment* property. Alignment takes effect only if the *WordWrap* property is *True*; if word wrapping is turned off, there is no margin to align to. For example, the following code attaches an *OnClick* event handler to a **Character ▶ Left** menu item, then attaches the same event handler to both a **Character ▶ Right** and **Character ▶ Center** menu item.
Sizing Owner-draw Items (see page 2167)	Before giving your application the chance to draw each item in a variable owner-draw control, the control receives a measure-item event, which is of type *TMeasureItemEvent*. *TMeasureItemEvent* tells the application where the item appears on the control. Delphi determines the size of the item (generally, it is just large enough to display the item's text in the current font). Your application can handle the event and change the rectangle chosen. For example, if you plan to substitute a bitmap for the item's text, change the rectangle to the size of the bitmap. If you want a bitmap and text,... more (see page 2167)

Starting a Drag Operation (see page 2168)	Every control has a property called *DragMode* that determines how drag operations are initiated. If *DragMode* is *dmAutomatic*, dragging begins automatically when the user presses a mouse button with the cursor on the control. Because *dmAutomatic* can interfere with normal mouse activity, you may want to set *DragMode* to *dmManual* (the default) and start the dragging by handling mouse-down events. To start dragging a control manually, call the control's *BeginDrag* method. *BeginDrag* takes a Boolean parameter called *Immediate* and, optionally, an integer parameter called *Threshold*. If you pass *True* for *Immediate*, dragging begins immediately. If you pass... more (see page 2168)
Working with Text in Controls (see page 2168)	The following topics how to use various features of rich edit and memo controls. Some of these features work with edit controls as well. • Setting text alignment (see page 2166) • Adding scrollbars at runtime (see page 2156) • Adding the clipboard object (see page 2156) • Selecting text (see page 2165) • Selecting all text (see page 2165) • Cutting (see page 2159) • Deleting selected text (see page 2160) • Disabling menu items (see page 2160) • Providing a pop-up menu (see page 2164) • Handling the OnPopup event (see page 2163)
Screen Scaling and ChangeScale (see page 2169)	The ChangeScale method is dependent on the BorderStyle property of forms. When BorderStyle is Single, you cannot resize the form manually, and ChangeScale adjusts form size as would be expected. However, if a form's BorderStyle is Sizeable, ChangeScale unexpectedly fails, as if expecting the user to take care of sizing. ChangeScale is sensitive to columns in a DBGrid that have been sized by the developer. ChangeScale typically fails on such columns. This can result in a DBGrid component and Form being adjusted by ChangeScale, but the Columns in the DBGrid not being resized. When implementing either of these methods, it... more (see page 2169)

3.2.4.12.1 Accepting Dragged Items

When the user drags something over a control, that control receives an *OnDragOver* event, at which time it must indicate whether it can accept the item if the user drops it there. The drag cursor changes to indicate whether the control can accept the dragged item. To accept items dragged over a control, attach an event handler to the control's *OnDragOver* event.

The drag-over event has a parameter called *Accept* that the event handler can set to *True* if it will accept the item. *Accept* changes the cursor type to an accept cursor or not.

The drag-over event has other parameters, including the source of the dragging and the current location of the mouse cursor, that the event handler can use to determine whether to accept the drag. In the following VCL example, a directory tree view accepts dragged items only if they come from a file list box.

```
procedure TFMForm.DirectoryOutline1DragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TFileListBox then
    Accept := True
  else
    Accept := False;
end;
void __fastcall TForm1::TreeView1DragOver(TObject *Sender, TObject *Source,
  int X, int Y, TDragState State, bool &Accept)
{
  if (Source->InheritsFrom(__classid(TFileListBox)))
```

```
}
```

See Also

TWinControl

Implementing Drag and Drop in Controls (see page 2160)

3.2.4.12.2 Adding Graphical Objects to a String List

Every string list has the ability to hold a list of objects in addition to its list of strings. You can also add graphical objects of varying sizes to a string list.

For example, in a file manager application, you may want to add bitmaps indicating the type of drive along with the letter of the drive. To do that, you need to add the bitmap images to the application, then copy those images into the proper places in the string list as described in the following sections.

Note that you can also organize graphical objects using an image list by creating a *TImageList*. However, these images must all be the same size. See Adding images to menu items (see page 1967) for an example of setting up an image list.

See Also

TStringList

TImage

3.2.4.12.3 Adding Graphics to Controls

Several controls let you customize the way the control is rendered. These include list boxes, combo boxes, menus, headers, tab controls, list views, status bars, tree views, and toolbars. Instead of using the standard method of drawing a control or its items, the control's owner (generally, the form) draws them at runtime. The most common use for owner-draw controls is to provide graphics instead of, or in addition to, text for items. For information on using owner-draw to add images to menus, see Adding images to menu items. (see page 1967).

All owner-draw controls contain lists of items. Usually, those lists are lists of strings that are displayed as text, or lists of objects that contain strings that are displayed as text. You can associate an object with each item in the list to make it easy to use that object when drawing items.

To create an owner-draw control:

1. Indicating that a control is owner-drawn. (see page 2166)
2. Adding graphical objects to a string list. (see page 2154)
3. Drawing owner-drawn items. (see page 2161)

See Also

Adding Images to Menu Items (see page 1967)

3.2.4.12.4 Adding Images to a String List

Once you have graphical images in an application, you can associate them with the strings in a string list. You can either add the objects at the same time as the strings, or associate objects with existing strings. The preferred method is to add objects and strings at the same time, if all the needed data is available.

The following example shows how you might want to add images to a string list. This is part of a file manager application where, along with a letter for each valid drive, it adds a bitmap indicating each drive's type. The *OnCreate* event handler looks like this:

```
procedure TFMForm.FormCreate(Sender: TObject);
```

```
var
  Drive: Char;
  AddedIndex: Integer;
begin
  for Drive := 'A' to 'Z' do  { iterate through all possible drives }
  begin
    case GetDriveType(Drive + ':/') of  { positive values mean valid drives }
     DRIVE_REMOVABLE:  { add a tab }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Floppy.Picture.Graphic);
     DRIVE_FIXED:  { add a tab }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Fixed.Picture.Graphic);
     DRIVE_REMOTE:  { add a tab }
       AddedIndex := DriveTabSet.Tabs.AddObject(Drive, Network.Picture.Graphic);
    end;
    if UpCase(Drive) = UpCase(DirectoryOutline.Drive) then  { current drive? }
     DriveTabSet.TabIndex := AddedIndex;  { then make that current tab }
  end;
end;
void __fastcall TFMForm::FormCreate(TObject *Sender)
{
  int AddedIndex;
  char DriveName[4] = "A:\\";
  for (char Drive = "A"; Drive <= "Z"; Drive++) // try all possible drives
  {
    DriveName[0] = Drive;
    switch (GetDriveType(DriveName))
    {
      case DRIVE_REMOVABLE:
        DriveName[1] = "\0"; // temporarily make drive letter into string
        AddedIndex = DriveList->Items->AddObject(DriveName,
           Floppy->Picture->Graphic);
        DriveName[1] = ":" // replace the colon
        break;
      case DRIVE_FIXED:
        DriveName[1] = "\0"; // temporarily make drive letter into string
        AddedIndex = DriveList->Items->AddObject(DriveName,
           Fixed->Picture->Graphic);
        DriveName[1] = ":" // replace the colon
        break;
      case DRIVE_REMOTE:
        DriveName[1] = "\0"; // temporarily make drive letter into string
        AddedIndex = DriveList->Items->AddObject(DriveName,
           Network->Picture->Graphic);
        DriveName[1] = ":" // replace the colon
        break;
    }
    if ((reinterpret_cast<int>(Drive - "A")) == getdisk()) // current drive?
      DriveList->ItemIndex = AddedIndex; // then make that the current list item
  }
}
```

See Also

TStringList

3.2.4.12.5 Adding Images to an Application

An image control is a nonvisual control that contains a graphical image, such as a bitmap. You use image controls to display graphical images on a form. You can also use them to hold hidden images that you'll use in your application. For example,

To store bitmaps for owner-draw controls in hidden image controls:

1. Add image controls to the main form.
2. Set their *Name* properties.

3. Set the Visible property for each image control to *False*.
4. Set the *Picture* property of each image to the desired bitmap using the Picture editor from the **Object Inspector**.

The image controls are invisible when you run the application. The image is stored with the form so it doesn't have to be loaded from a file at runtime.

See Also

TImage

3.2.4.12.6 Adding Scroll Bars at Runtime

Rich edit and memo components can contain horizontal or vertical scroll bars, or both, as needed. When word wrapping is enabled, the component needs only a vertical scroll bar. If the user turns off word wrapping, the component might also need a horizontal scroll bar, since text is not limited by the right side of the editor.

To add scroll bars at runtime:

1. Determine whether the text might exceed the right margin. In most cases, this means checking whether word wrapping is enabled. You might also check whether any text lines actually exceed the width of the control.
2. Set the rich edit or memo component's *ScrollBars* property to include or exclude scroll bars.

The following example attaches an *OnClick* event handler to a **Character ▸ WordWrap** menu item.

```
procedure TForm.WordWrap1Click(Sender: TObject);
begin
  with Editor do
  begin
    WordWrap := not WordWrap;  { toggle word wrapping }
    if WordWrap then
      ScrollBars := ssVertical  { wrapped requires only vertical }
    else
      ScrollBars := ssBoth;  { unwrapped might need both }
      WordWrap1.Checked := WordWrap;  { check menu item to match property }
  end;
end;
void __fastcall TForm::WordWrap1Click(TObject *Sender)
{
  Editor->WordWrap = !(Editor->WordWrap);   // toggle word wrapping
  if (Editor->WordWrap)
    Editor->ScrollBars = ssVertical;        // wrapped requires only vertical
  else
    Editor->ScrollBars = ssBoth;            // unwrapped can need both
  WordWrap1->Checked = Editor->WordWrap;    // check menu item to match property
}
```

The rich edit and memo components handle their scroll bars in a slightly different way. The rich edit component can hide its scroll bars if the text fits inside the bounds of the component. The memo always shows scroll bars if they are enabled.

See Also

TMemo

TRichEdit

3.2.4.12.7 Adding the Clipboard Object

Most text-handling applications provide users with a way to move selected text between documents, including documents in different applications. *TClipboard* object encapsulates a clipboard (such as the Windows Clipboard) and includes methods for cutting, copying, and pasting text (and other formats, including graphics). The *Clipboard* object is declared in the *Clipbrd* unit.

To add the Clipboard object to an application:

1. Select the unit that will use the clipboard.
2. Search for the **implementation** reserved word.
3. Add *Clipbrd* to the **uses** clause below **implementation**.

- If there is already a **uses** clause in the **implementation** part, add *Clipbrd* to the end of it.
- If there is not already a **uses** clause, add one that says

```
    uses Clipbrd;
#include <vcl\Clipbrd.hpp>
```

For example, in an application with a child window, the uses clause in the unit's implementation part might look like this:

```
uses
  MDIFrame, Clipbrd;
```

See Also

TClipboard

3.2.4.12.8 Changing the Drag Mouse Pointer

You can customize the appearance of the mouse pointer during drag operations by setting the source component's *DragCursor* property (VCL only).

See Also

TWinControl

3.2.4.12.9 Controlling How Child Controls Are Docked

A docking site automatically accepts child controls when they are released over the docking site. For most controls, the first child is docked to fill the client area, the second splits that into separate regions, and so on. Page controls dock children into new tab sheets (or merge in the tab sheets if the child is another page control).

Three events allow docking sites to further constrain how child controls are docked:

```
property OnGetSiteInfo: TGetSiteInfoEvent;
TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl; var   InfluenceRect:
TRect; var CanDock: Boolean) of object;
__property TGetSiteInfoEvent OnGetSiteInfo = {read=FOnGetSiteInfo, write=FOnGetSiteInfo};
typedef void __fastcall (__closure *TGetSiteInfoEvent)(System::TObject* Sender, TControl*
DockClient, Windows::TRect &InfluenceRect, const Windows::TPoint &MousePos, bool &CanDock);
```

OnGetSiteInfo occurs on the docking site when the user drags a dockable child over the control. It allows the site to indicate whether it will accept the control specified by the *DockClient* parameter as a child, and if so, where the child must be to be considered for docking. When *OnGetSiteInfo* occurs, *InfluenceRect* is initialized to the screen coordinates of the docking site, and *CanDock* is initialized to *True*. A more limited docking region can be created by changing *InfluenceRect* and the child can be rejected by setting *CanDock* to *False*.

```
property OnDockOver: TDockOverEvent;
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer; State:
TDragState; var Accept: Boolean) of object;
__property TDockOverEvent OnDockOver = {read=FOnDockOver, write=FOnDockOver};
typedef void __fastcall (__closure *TDockOverEvent)(System::TObject* Sender, TDragDockObject*
Source, int X, int Y, TDragState State, bool &Accept);
```

OnDockOver occurs on the docking site when the user drags a dockable child over the control. It is analogous to the *OnDragOver* event in a drag-and-drop operation. Use it to signal that the child can be released for docking, by setting the *Accept*

parameter. If the dockable control is rejected by the *OnGetSiteInfo* event handler (perhaps because it is the wrong type of control), *OnDockOver* does not occur.

```
property OnDockDrop: TDockDropEvent;
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X, Y:  Integer) of object;
__property TDockDropEvent OnDockDrop = {read=FOnDockDrop, write=FOnDockDrop};
typedef void __fastcall (__closure *TDockDropEvent)(System::TObject* Sender, TDragDockObject*
Source, int X, int Y);
```

OnDockDrop occurs on the docking site when the user releases the dockable child over the control. It is analogous to the *OnDragDrop* event in a normal drag-and-drop operation. Use this event to perform any necessary accommodations to accepting the control as a child control. Access to the child control can be obtained using the *Control* property of the *TDockObject* specified by the *Source* parameter.

See Also

TWinControl

TControl

3.2.4.12.10 Controlling How Child Controls Are Undocked

A docking site automatically allows child controls to be undocked when they are dragged and have a *DragMode* property of *dmAutomatic*. Docking sites can respond when child controls are dragged off, and even prevent the undocking, in an *OnUnDock* event handler:

```
property OnUnDock: TUnDockEvent;
TUnDockEvent = procedure(Sender: TObject; Client: TControl; var Allow: Boolean)   of object;
__property TUnDockEvent OnUnDock = {read=FOnUnDock, write=FOnUnDock};
typedef void __fastcall (__closure *TUnDockEvent)(System::TObject* Sender, TControl* Client,
TWinControl* NewTarget, bool &Allow);
```

The *Client* parameter indicates the child control that is trying to undock, and the *Allow* parameter lets the docking site (*Sender*) reject the undocking. When implementing an *OnUnDock* event handler, it can be useful to know what other children (if any) are currently docked. This information is available in the read-only *DockClients* property, which is an indexed array of *TControl*. The number of dock clients is given by the read-only *DockClientCount* property.

See Also

TWinControl

TControl

3.2.4.12.11 Controlling How Child Controls Respond to Drag-and-dock Operations

Dockable child controls have two events that occur during drag-and-dock operations: *OnStartDock*, analogous to the *OnStartDrag* event of a drag-and-drop operation, allows the dockable child control to create a custom drag object. *OnEndDock*, like *OnEndDrag*, occurs when the dragging terminates.

See Also

TWinControl

TControl

3.2.4.12.12 Customizing Drag and Drop with a Drag Object

You can use a *TDragObject* descendant to customize an object's drag-and-drop behavior. The standard drag-over and

drag-and-drop events indicate the source of the dragged item and the coordinates of the mouse cursor over the accepting control. To get additional state information, derive a custom drag object from *TDragObject* or *TDragObjectEx* (VCL only) and override its virtual methods. Create the custom drag object in the *OnStartDrag* event.

Normally, the source parameter of the drag-over and drag-and-drop events is the control that starts the drag operation. If different kinds of control can start an operation involving the same kind of data, the source needs to support each kind of control. When you use a descendant of *TDragObject*, however, the source is the drag object itself; if each control creates the same kind of drag object in its *OnStartDrag* event, the target needs to handle only one kind of object. The drag-over and drag-and-drop events can tell if the source is a drag object, as opposed to the control, by calling the *IsDragObject* function.

TDragObjectEx descendants (VCL only) are freed automatically whereas descendants of *TDragObject* are not. If you have *TDragObject* descendants that you are not explicitly freeing, you can change them so they descend from *TDragObjectEx* instead to prevent memory loss.

Drag objects let you drag items between a form implemented in the application's main executable file and a form implemented using a DLL, or between forms that are implemented using different DLLs.

See Also

TWinControl

TDragObject

IsDragObjectfunction

3.2.4.12.13 Cutting, Copying, and Pasting Text

Applications that use the *Clipbrd* unit can cut, copy, and paste text, graphics, and objects through the clipboard. The edit components that encapsulate the standard text-handling controls all have methods built into them for interacting with the clipboard.

To cut, copy, or paste text with the clipboard, call the edit component's *CutToClipboard*, *CopyToClipboard*, and *PasteFromClipboard* methods, respectively.

For example, the following code attaches event handlers to the *OnClick* events of the **Edit ▸ Cut**, **Edit ▸ Copy**, and **Edit ▸ Paste** commands, respectively:

```
procedure TEditForm.CutToClipboard(Sender: TObject);
begin
  Editor.CutToClipboard;
end;
procedure TEditForm.CopyToClipboard(Sender: TObject);
begin
  Editor.CopyToClipboard;
end;
procedure TEditForm.PasteFromClipboard(Sender: TObject);
begin
  Editor.PasteFromClipboard;
end;
void __fastcall TMainForm::EditCutClick(TObject* Sender)
{    RichEdit1->CutToClipboard();
}
void __fastcall TMainForm::EditCopyClick(TObject* Sender)
{    RichEdit1->CopyToClipboard();
}
void __fastcall TMainForm::EditPasteClick(TObject* Sender)
{    RichEdit1->PasteFromClipboard();
}
```

See Also

Using the Clipboard with Graphics (see page 2206)

TClipboard

3.2.4.12.14 Deleting Selected Text

You can delete the selected text in an edit component without cutting it to the clipboard. To do so, call the *ClearSelection* method. For example, if you have a Delete item on the Edit menu, your code could look like this:

```
procedure TEditForm.Delete(Sender: TObject);
begin
  RichEdit1.ClearSelection;
end;
void __fastcall TMainForm::EditDeleteClick(TObject *Sender)
{
  RichEdit1->ClearSelection();
}
```

3.2.4.12.15 Disabling Menu Items

It is often useful to disable menu commands without removing them from the menu. For example, in a text editor, if there is no text currently selected, the Cut, Copy, and Delete commands are inapplicable. An appropriate time to enable or disable menu items is when the user selects the menu. To disable a menu item, set its *Enabled* property to *False*.

In the following example, an event handler is attached to the *OnClick* event for the Edit item on a child form's menu bar. It sets *Enabled* for the Cut, Copy, and Delete menu items on the Edit menu based on whether *RichEdit1* has selected text. The Paste command is enabled or disabled based on whether any text exists on the clipboard.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;  { declare a temporary variable }
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);  {enable or disable the
Paste                                                menu item}
  HasSelection := Editor.SelLength > 0;  { True if text is selected }
  Cut1.Enabled := HasSelection;  { enable menu items if HasSelection is True }
  Copy1.Enabled := HasSelection;
  Delete1.Enabled := HasSelection;
end;
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // enable or disable the Paste menu item
  Paste1->Enabled = Clipboard()->HasFormat(CF_TEXT);
  bool HasSelection = (RichEdit1->SelLength > 0);  // true if text is selected
  Cut1->Enabled = HasSelection;  // enable menu items if HasSelection is true
  Copy1->Enabled = HasSelection;
  Delete1->Enabled = HasSelection;
}
```

The *HasFormat* method of the clipboard returns a Boolean value based on whether the clipboard contains objects, text, or images of a particular format. By calling *HasFormat* with the parameter *CF_TEXT*, you can determine whether the clipboard contains any text, and enable or disable the Paste item as appropriate.

See Also

TMenu

3.2.4.12.16 Implementing Drag and Drop in Controls

Drag-and-drop is often a convenient way for users to manipulate objects. You can let users drag an entire control, or let them drag items from one control—such as a list box or tree view—into another.

- Starting a drag operation (see page 2168)
- Accepting dragged items (see page 2153)
- Dropping items (see page 2162)
- Ending a drag operation (see page 2162)
- Customizing drag and drop with a drag object (see page 2158)
- Changing the drag mouse pointer (see page 2157)

See Also

TWinControl

TWidgetControl

3.2.4.12.17 Drawing Owner-draw Items

When an application needs to draw or redraw an owner-draw control, the operating system generates draw-item events for each visible item in the control. Depending on the control, the item may also receive draw events for the item as a part of the item.

To draw each item in an owner-draw control, attach an event handler to the draw-item event for that control.

The names of events for owner drawing typically start with one of the following:

- *OnDraw*, such as *OnDrawItem* or *OnDrawCell*
- *OnCustomDraw*, such as *OnCustomDrawItem*
- *OnAdvancedCustomDraw*, such as *OnAdvancedCustomDrawItem*

The draw-item event contains parameters identifying the item to draw, the rectangle in which to draw, and usually some information about the state of the item (such as whether the item has focus). The application handles each event by rendering the appropriate item in the given rectangle.

For example, the following code shows how to draw items in a list box that has bitmaps associated with each string. It attaches this handler to the *OnDrawItem* event for the list box:

```
procedure TFMForm.DriveTabSetDrawTab(Sender: TObject; TabCanvas: TCanvas;
  R: TRect; Index: Integer; Selected: Boolean);
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap(DriveTabSet.Tabs.Objects[Index]);
  with TabCanvas do
  begin
     Draw(R.Left, R.Top + 4, Bitmap);  { draw bitmap }
     TextOut(R.Left + 2 + Bitmap.Width,  { position text }
     R.Top + 2, DriveTabSet.Tabs[Index]);  { and draw it to the right of the bitmap }
  end;
end;
void __fastcall TForm1::ListBox1DrawItem(TWinControl *Control, int Index,
  TRect &Rect, TOwnerDrawState State)
  TBitmap *Bitmap = (TBitmap *)ListBox1->Items->Objects[Index];
  ListBox1->Canvas->Draw(R.Left, R.Top + 2, Bitmap); // draw the bitmap
  ListBox1->Canvas->TextOut(R.Left + Bitmap->Width + 2, R.Top + 2,
    ListBox1->Items->Strings[Index]);        // and write the text to its right
}
```

3.2.4.12.18 Drawing Owner-drawn Items

When you indicate that a control is owner-drawn, either by setting a property or supplying a custom draw event handler, the control is no longer drawn on the screen. Instead, the operating system generates events for each visible item in the control. Your application handles the events to draw the items.

To draw the items in an owner-draw control, do the following for each visible item in the control. Use a single event handler for all items.

1. Size the item, if needed. Items of the same size (for example, with a list box style of *lsOwnerDrawFixed*), do not require sizing.
2. Draw the item.

3.2.4.12.19 Dropping Items

If a control indicates that it can accept a dragged item, it needs to handle the item should it be dropped. To handle dropped items, attach an event handler to the *OnDragDrop* event of the control accepting the drop. Like the drag-over event, the drag-and-drop event indicates the source of the dragged item and the coordinates of the mouse cursor over the accepting control. The latter parameter allows you to monitor the path an item takes while being dragged; you might, for example, want to use this information to change the color of components if an item is dropped.

In the following VCL example, a directory tree view, accepting items dragged from a file list box, responds by moving files to the directory on which they are dropped.

```
procedure TFMForm.DirectoryOutline1DragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  if Source is TFileListBox then
    with DirectoryOutline1 do
      ConfirmChange('Move', FileListBox1.FileName, Items[GetItem(X, Y)].FullPath);
end;
void __fastcall TForm1::TreeView1DragDrop(TObject *Sender, TObject *Source,
  if (Source->InheritsFrom(__classid(TFileListBox)))
  {
    TTreeNode *pNode = TreeView1->GetNodeAt(X,Y); // pNode is drop target
    AnsiString NewFile = pNode->Text + AnsiString("//") +
      ExtractFileName(FileListBox1->FileName); // build file name for drop target
    MoveFileEx(FileListBox1->FileName.c_str(), NewFile.c_str(),
      MOVEFILE_REPLACE_EXISTING | MOVEFILE_COPY_ALLOWED); // move the file
  }
}
```

See Also

TWinControl

Implementing Drag and Drop in Controls (see page 2160)

3.2.4.12.20 Ending a Drag Operation

A drag operation ends when the item is either successfully dropped or released over a control that cannot accept it. At this point an end-drag event is sent to the control from which the drag was initiated. To enable a control to respond when items have been dragged from it, attach an event handler to the control's *OnEndDrag* event.

The most important parameter in an *OnEndDrag* event is called *Target*, which indicates which control, if any, accepts the drop. If *Target* is **nil**, it means no control accepts the dragged item. The *OnEndDrag* event also includes the coordinates on the receiving control.

In the following VCL example, a file list box handles an end-drag event by refreshing its file list.

```
procedure TFMForm.FileListBox1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
  if Target <> nil then FileListBox1.Update;
end;
void __fastcall TFMForm::FileListBox1EndDrag(TObject *Sender, TObject *Target,   int X, int Y)
  if (Target)
    FileListBox1->Update();
```

```
};
```

See Also

TWinControl

Implementing Drag and Drop in Controls (see page 2160)

3.2.4.12.21 Handling the OnPopup Event

You may want to adjust pop-up menu items before displaying the menu, just as you may want to enable or disable items on a regular menu. With a regular menu, you can handle the *OnClick* event for the item at the top of the menu.

With a pop-up menu, however, there is no top-level menu bar, so to prepare the pop-up menu commands, you handle the event in the menu component itself. The pop-up menu component provides an event just for this purpose, called *OnPopup*.

To adjust menu items on a pop-up menu before displaying them:

1. Select the pop-up menu component.
2. Attach an event handler to its *OnPopup* event.
3. Write code in the event handler to enable, disable, hide, or show menu items.

In the following code, the *Edit1Click* event handler described previously in Disabling menu items (see page 2160) is attached to the pop-up menu component's OnPopup event. A line of code is added to *Edit1Click* for each item in the pop-up menu.

```
procedure TEditForm.Edit1Click(Sender: TObject);
var
  HasSelection: Boolean;
begin
  Paste1.Enabled := Clipboard.HasFormat(CF_TEXT);
  Paste2.Enabled := Paste1.Enabled;{Add this line}
  HasSelection := Editor.SelLength <> 0;
  Cut1.Enabled := HasSelection;
  Cut2.Enabled := HasSelection;{Add this line}
  Copy1.Enabled := HasSelection;
  Copy2.Enabled := HasSelection;{Add this line}
  Delete1.Enabled := HasSelection;
end;
void __fastcall TMainForm::EditEditClick(TObject *Sender)
{
  // enable or disable the Paste menu item
  Paste1->Enabled = Clipboard()->HasFormat(CF_TEXT);
  Paste2->Enabled = Paste1->Enabled;            // add this line
  bool HasSelection = (RichEdit1->SelLength > 0);  // true if text is selected
  Cut1->Enabled = HasSelection;    // enable menu items if HasSelection is true
  Cut2->Enabled = HasSelection;    // add this line
  Copy1->Enabled = HasSelection;
  Copy2->Enabled = HasSelection;   // add this line
  Delete1->Enabled = HasSelection;
}
```

3.2.4.12.22 Implementing Drag and Dock in Controls

Descendants of *TWinControl* can act as docking sites and descendants of *TControl* can act as child windows that are docked into docking sites. For example, to provide a docking site at the left edge of a form window, align a panel to the left edge of the form and make the panel a docking site. When dockable controls are dragged to the panel and released, they become child controls of the panel.

- Making a windowed control a docking site (see page 2164)
- Making a control a dockable child (see page 2164)

- Controlling how child controls are docked (see page 2157)
- Controlling how child controls are undocked (see page 2158)
- Controlling how child controls respond to drag-and-dock operations (see page 2158)

See Also

TWinControl

TControl

3.2.4.12.23 Making a Control a Dockable Child

To make a control a dockable child:

1. Set its *DragKind* property to *dkDock*. When *DragKind* is *dkDock*, dragging the control moves the control to a new docking site or undocks the control so that it becomes a floating window. When *DragKind* is *dkDrag* (the default), dragging the control starts a drag-and-drop operation which must be implemented using the *OnDragOver*, *OnEndDrag*, and *OnDragDrop* events.
2. Set its *DragMode* to *dmAutomatic*. When *DragMode* is *dmAutomatic*, dragging (for drag-and-drop or docking, depending on *DragKind*) is initiated automatically when the user starts dragging the control with the mouse. When *DragMode* is *dmManual*, you can still begin a drag-and-dock (or drag-and-drop) operation by calling the *BeginDrag* method.
3. Set its *FloatingDockSiteClass* property to indicate the *TWinControl* descendant that should host the control when it is undocked and left as a floating window. When the control is released and not over a docking site, a windowed control of this class is created dynamically, and becomes the parent of the dockable child. If the dockable child control is a descendant of *TWinControl*, it is not necessary to create a separate floating dock site to host the control, although you may want to specify a form in order to get a border and title bar. To omit a dynamic container window, set *FloatingDockSiteClass* to the same class as the control, and it will become a floating window with no parent.

See Also

TWinControl

TControl

3.2.4.12.24 Making a Windowed Control a Docking Site

To make a windowed control a docking site:

1. Set the *DockSite* property to *True*.
2. If the dock site object should not appear except when it contains a docked client, set its *AutoSize* property to *True*. When *AutoSize* is *True*, the dock site is sized to 0 until it accepts a child control for docking. Then it resizes to fit around the child control.

See Also

TWinControl

TControl

3.2.4.12.25 Providing a Pop-up Menu

Pop-up, or local, menus are a common ease-of-use feature for any application. They enable users to minimize mouse movement by clicking the right mouse button in the application workspace to access a list of frequently used commands.

In a text editor application, for example, you can add a pop-up menu that repeats the Cut, Copy, and Paste editing commands. These pop-up menu items can use the same event handlers as the corresponding items on the Edit menu. You don't need to create accelerator or shortcut keys for pop-up menus because the corresponding regular menu items generally already have shortcuts.

A form's *PopupMenu* property specifies what pop-up menu to display when a user right-clicks any item on the form. Individual controls also have *PopupMenu* properties that can override the form's property, allowing customized menus for particular controls.

To add a pop-up menu to a form:

1. Place a pop-up menu component on the form.
2. Use the Menu Designer to define the items for the pop-up menu.
3. Set the *PopupMenu* property of the form or control that displays the menu to the name of the pop-up menu component.
4. Attach handlers to the *OnClick* events of the pop-up menu items.

See Also

TPopupMenu

Handling the OnPopup Event (see page 2163)

3.2.4.12.26 Selecting All Text

The *SelectAll* method selects the entire contents of an edit control, such as a rich edit or memo component. This is especially useful when the component's contents exceed the visible area of the component. In most other cases, users select text with either keystrokes or mouse dragging.

To select the entire contents of a rich edit or memo control, call the *RichEdit1* control's *SelectAll* method.

For example:

```
procedure TMainForm.SelectAll(Sender: TObject);
begin
  RichEdit1.SelectAll;  { select all text in RichEdit }
end;
void __fastcall TMainForm::SelectAll(TObject *Sender)
{
  RichEdit1->SelectAll();     // select all text in RichEdit
}
```

3.2.4.12.27 Selecting Text

For text in an edit control, before you can send any text to the clipboard, that text must be selected. Highlighting of selected text is built into the edit components. When the user selects text, it appears highlighted.

The table below lists properties commonly used to handle selected text.

Properties of selected text

Property	Description
SelText	Contains a string representing the selected text in the component.
SelLength	Contains the length of a selected string.
SelStart	Contains the starting position of a string relative to the beginning of an edit control's text buffer.

For example, the following *OnFind* event handler searches a Memo component for the text specified in the *FindText* property of a find dialog component. If found, the first occurrence of the text in Memo1 is selected.

```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
I, J, PosReturn, SkipChars: Integer;
begin
for I := 0 to Memo1.Lines.Count do
```

```
begin
PosReturn := Pos(FindDialog1.FindText,Memo1.Lines[I]);
if PosReturn <> 0 then {found!}
begin
Skipchars := 0;
for J := 0 to I - 1 do
Skipchars := Skipchars + Length(Memo1.Lines[J]);
SkipChars := SkipChars + (I*2);
SkipChars := SkipChars + PosReturn - 1;
      Memo1.SetFocus;
Memo1.SelStart := SkipChars;
Memo1.SelLength := Length(FindDialog1.FindText);
Break;
end;
end;
end;
```

See Also

TClipboard

3.2.4.12.28 Indicating That a Control Is Owner-drawn

To customize the drawing of a control, you must supply event handlers that render the control's image when it needs to be painted. Some controls receive these events automatically. For example, list views, tree views, and toolbars all receive events at various stages in the drawing process without your having to set any properties. These events have names such as *OnCustomDraw* or *OnAdvancedCustomDraw*.

Other controls, however, require you to set a property before they receive owner-draw events. List boxes, combo boxes, header controls, and status bars have a property called *Style*. *Style* determines whether the control uses the default drawing (called the "standard" style) or owner drawing. Grids use a property called *DefaultDrawing* to enable or disable the default drawing. List views and tab controls have a property called *OwnerDraw* that enables or disabled the default drawing.

List boxes and combo boxes have additional owner-draw styles, called fixed and *variable*, as the following table describes. Other controls are always fixed, although the size of the item that contains the text may vary, the size of each item is determined before drawing the control.

Fixed vs. variable owner-draw styles

Owner-draw style	Meaning	Examples
Fixed	Each item is the same height, with that height determined by the ItemHeight property.	*lbOwnerDrawFixed*,*csOwnerDrawFixed*
Variable	Each item might have a different height, determined by the data at runtime.	*lbOwnerDrawVariable*, *csOwnerDrawVariable*

3.2.4.12.29 Setting Text Alignment

In a rich edit or memo component, text can be left- or right-aligned or centered. To change text alignment, set the edit component's *Alignment* property. Alignment takes effect only if the *WordWrap* property is *True*; if word wrapping is turned off, there is no margin to align to.

For example, the following code attaches an *OnClick* event handler to a **Character ▸ Left** menu item, then attaches the same event handler to both a **Character ▸ Right** and **Character ▸ Center** menu item.

```
procedure TForm.AlignClick(Sender: TObject);
begin
  Left1.Checked := False;  { clear all three checks }
```

```
  Right1.Checked := False;
  Center1.Checked := False;
  with Sender as TMenuItem do Checked := True;  { check the item clicked }
  with Editor do  { then set Alignment to match }
    if Left1.Checked then
      Alignment := taLeftJustify
    else if Right1.Checked then
      Alignment := taRightJustify
    else if Center1.Checked then
      Alignment := taCenter;
end;
```

```
switch(reinterpret_cast<int>(RichEdit1->Paragraph->Alignment))
{
  case 0: LeftAlign->Down   = true; break;
  case 1: RightAlign->Down  = true; break;
  case 2: CenterAlign->Down = true; break;
}
```

You can also use the *HMargin* property to adjust the left and right margins in a memo control.

See Also

TMemo

TRichEdit

3.2.4.12.30 Sizing Owner-draw Items

Before giving your application the chance to draw each item in a variable owner-draw control, the control receives a measure-item event, which is of type *TMeasureItemEvent*. *TMeasureItemEvent* tells the application where the item appears on the control.

Delphi determines the size of the item (generally, it is just large enough to display the item's text in the current font). Your application can handle the event and change the rectangle chosen. For example, if you plan to substitute a bitmap for the item's text, change the rectangle to the size of the bitmap. If you want a bitmap and text, adjust the rectangle to be large enough for both.

To change the size of an owner-draw item, attach an event handler to the measure-item event in the owner-draw control. Depending on the control, the name of the event can vary. List boxes and combo boxes use *OnMeasureItem*. Grids have no measure-item event.

The sizing event has two important parameters: the index number of the item and the height of that item. The height is variable: the application can make it either smaller or larger. The positions of subsequent items depend on the size of preceding items.

For example, in a variable owner-draw list box, if the application sets the height of the first item to five pixels, the second item starts at the sixth pixel down from the top, and so on. In list boxes and combo boxes, the only aspect of the item the application can alter is the height of the item. The width of the item is always the width of the control.

Owner-draw grids cannot change the sizes of their cells as they draw. The size of each row and column is set before drawing by the *ColWidths* and *RowHeights* properties.

The following code, attached to the *OnMeasureItem* event of an owner-draw list box, increases the height of each list item to accommodate its associated bitmap.

```
procedure TFMForm.ListBox1MeasureItem(Control: TWinControl; Index: Integer;
  var Height: Integer);  { note that Height is a var parameter}
var
  BitmapHeight: Integer;
begin
  BitmapHeight := TBitmap(ListBox1.Items.Objects[Index]).Height;
  { make sure the item height has enough room, plus two }
  Height := Max(Height, Bitmap Height +2);
```

```
end;
void __fastcall TForm1::ListBox1MeasureItem(TWinControl *Control, int Index,
  int &Height)      // note that Height is passed by reference
{
  int BitmapHeight = (dynamic_cast<TBitmap *>(ListBox1->Items->Objects[Index]))->Height + 2;
  // make sure list item has enough room for bitmap (plus two)
  Height = Max(Height, Bitmap Height +2);
}
```

Note: You must typecast the items from the *Objects* property in the string list. *Objects* is a property of type *TObject* so that it can hold any kind of object. When you retrieve objects from the array, you need to typecast them back to the actual type of the items.

3.2.4.12.31 Starting a Drag Operation

Every control has a property called *DragMode* that determines how drag operations are initiated. If *DragMode* is *dmAutomatic*, dragging begins automatically when the user presses a mouse button with the cursor on the control. Because *dmAutomatic* can interfere with normal mouse activity, you may want to set *DragMode* to *dmManual* (the default) and start the dragging by handling mouse-down events.

To start dragging a control manually, call the control's *BeginDrag* method. *BeginDrag* takes a Boolean parameter called *Immediate* and, optionally, an integer parameter called *Threshold*. If you pass *True* for *Immediate*, dragging begins immediately. If you pass *False*, dragging does not begin until the user moves the mouse the number of pixels specified by *Threshold*. Calling

```
BeginDrag (False);
false, -1
```

allows the control to accept mouse clicks without beginning a drag operation.

You can place other conditions on whether to begin dragging, such as checking which mouse button the user pressed, by testing the parameters of the mouse-down event before calling *BeginDrag*. The following code, for example, handles a mouse-down event in a file list box by initiating a drag operation only if the left mouse button was pressed.

```
procedure TFMForm.FileListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft then  { drag only if left button pressed }
    with Sender as TFileListBox do  { treat Sender as TFileListBox }
      begin
        if ItemAtPos(Point(X, Y), True) >= 0 then  { is there an item here? }
          BeginDrag(False);  { if so, drag it }
    end;
end;
void __fastcall TFMForm::FileListBox1MouseDown(TObject *Sender,
{
  if (Button == mbLeft)
  {
    TFileListBox *pLB = dynamic_cast<TFileListBox *>(Sender); // cast to TFileListBox
    if (pLB && pLB->ItemAtPos(Point(X,Y), true) >= 0) // is there an item here?
      pLB->BeginDrag(false, -1);                      // if so, drag it
  }
}
```

See Also

TWinControl

Implementing Drag and Drop in Controls (see page 2160)

3.2.4.12.32 Working with Text in Controls

The following topics how to use various features of rich edit and memo controls. Some of these features work with edit controls as well.

- Setting text alignment (see page 2166)
- Adding scrollbars at runtime (see page 2156)
- Adding the clipboard object (see page 2156)
- Selecting text (see page 2165)
- Selecting all text (see page 2165)
- Cutting (see page 2159)
- Deleting selected text (see page 2160)
- Disabling menu items (see page 2160)
- Providing a pop-up menu (see page 2164)
- Handling the OnPopup event (see page 2163)

3.2.4.12.33 Screen Scaling and ChangeScale

The ChangeScale method is dependent on the BorderStyle property of forms. When BorderStyle is Single, you cannot resize the form manually, and ChangeScale adjusts form size as would be expected. However, if a form's BorderStyle is Sizeable, ChangeScale unexpectedly fails, as if expecting the user to take care of sizing.

ChangeScale is sensitive to columns in a DBGrid that have been sized by the developer. ChangeScale typically fails on such columns. This can result in a DBGrid component and Form being adjusted by ChangeScale, but the Columns in the DBGrid not being resized.

When implementing either of these methods, it is relatively easy to perform the logical conversions from the built-in variant types. You must consider, however, the possibility that the variant to or from which you are casting may be another custom Variant type. To handle this situation, you can try casting to one of the built-in Variant types as an intermediate step.

You cannot remove the column sizing through the IDE. In Form Text mode, you need to remove the Width statement on the column definition.

See Also

ChangeScale

TControl Branch (see page 2060)

3.2.4.13 Working with graphics and multimedia

Topics

Name	Description
Changing the Brush Style (see page 2175)	A brush style determines what pattern the canvas uses to fill shapes. It lets you specify various ways to combine the brush's color with any colors already on the canvas. The predefined styles include solid color, no color, and various line and hatch patterns. To change the style of a brush, set its Style property to one of the predefined values: bsBDiagonal, bsClear, bsCross, bsDiagCross, bsFDiagonal, bsHorizontal, bsSolid, or bsVertical. Cross-platform applications include the predefined values of bsDense1 through bsDense7. This example sets brush styles by sharing a click-event handler for a set of eight brush-style buttons. All eight buttons... more (see page 2175)

Overview of Graphics Programming (see page 2176)	In VCL applications, the graphics components defined in the Graphics unit encapsulate the Windows Graphics Device Interface (GDI), making it easy to add graphics to your Windows applications. To draw graphics in an application, you draw on an object's *canvas*, rather than directly on the object. The canvas is a property of the object, and is itself an object. A main advantage of the canvas object is that it handles resources effectively and it manages the device context for you, so your programs can use the same methods regardless of whether you are drawing on the screen,... more (see page 2176)
Working with Multimedia (see page 2177)	You can add multimedia components to your applications. To do this, you can use either the TAnimate component on the Win32 page or the TMediaPlayer component on the System category of the **Tool palette**. Use the animate component when you want to add silent video clips to your application. Use the media player component when you want to add audio and/or video clips to an application. This topic discusses: • Adding silent video clips to an application (see page 2179) • Adding audio and/or video clips to an application (see page 2177)
Adding Audio and/or Video Clips to an Application (see page 2177)	With the media player component, you can add audio and/or video clips to your application. It opens a media device and plays, stops, pauses, records, etc., the audio and/or video clips used by the media device. The media device may be hardware or software. **Note:** Audio support is not available in cross-platform applications.
Adding Silent Video Clips to an Application (see page 2179)	With the animation control, you can add silent video clips to your application:
Example of Adding Audio and/or Video Clips (VCL Only) (see page 2179)	This example runs an AVI video clip of a multimedia advertisement.
Example of Adding Silent Video Clips (see page 2180)	Suppose you want to display an animated logo as the first screen that appears when your application starts. After the logo finishes playing the screen disappears.
Adding a Field to a Form Object to Track Mouse Actions (see page 2181)	To track whether a mouse button was pressed, you must add an object field to the form object. When you add a component to a form, Delphi adds a field that represents that component to the form object, so that you can refer to the component by the name of its field. You can also add your own fields to forms by editing the type declaration in the form unit's header file. In the following example, the form needs to track whether the user has pressed a mouse button. To do that, it adds a Boolean field and sets its... more (see page 2181)
Adding an Image Control (see page 2181)	An image control is a container component that allows you to display your bitmap objects. You use an image control to hold a bitmap that is not necessarily displayed all the time, or which an application needs to use to generate other pictures. **Note:** Adding Graphics to Controls (see page 2154) shows how to use graphics in controls.
Changing the Brush Color (see page 2181)	A brush's color determines what color the canvas uses to fill shapes. To change the fill color, assign a value to the brush's *Color* property. Brush is used for background color in text and line drawing so you typically set the background color property. You can set the brush color just as you do the pen color, in response to a click on a color grid on the brush's toolbar :
Changing the Pen Color (see page 2182)	You can set the color of a pen as you would any other *Color* property at runtime. A pen's color determines the color of the lines the pen draws, including lines drawn as the boundaries of shapes, as well as other lines and polylines. To change the pen color, assign a value to the *Color* property of the pen. To let the user choose a new color for the pen, put a color grid on the pen's toolbar. A color grid can set both foreground and background colors. For a non-grid pen style, you must consider the background color, which... more (see page 2182)
Changing the Pen Mode (see page 2182)	A pen's *Mode* property lets you specify various ways to combine the pen's color with the color on the canvas. For example, the pen could always be black, be an inverse of the canvas background color, inverse of the pen color, and so on.
Changing the Pen Style (see page 2182)	A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on. The task of setting the properties of pen is an ideal case for having different controls share same event handler to handle events. To determine which control actually got the event, you check the *Sender* parameter.

Changing the Pen Width (see page 2183)	A pen's width determines the thickness, in pixels, of the lines it draws. **Note:** When the thickness is greater than 1, Windows always draws solid lines, regardless of the value of the pen's *Style* property. To change the pen width, assign a numeric value to the pen's Width property. Suppose you have a scroll bar on the pen's toolbar to set width values for the pen. And suppose you want to update the label next to the scroll bar to provide feedback to the user. Using the scroll bar's position to determine the pen width, you update the pen width... more (see page 2183)
Changing the Tool with Speed Buttons (see page 2184)	Each drawing tool needs an associated *OnClick* event handler. Suppose your application had a toolbar button for each of four drawing tools: line, rectangle, ellipse, and rounded rectangle. You would attach the following event handlers to the *OnClick* events of the four drawing-tool buttons, setting *DrawingTool* to the appropriate value for each:
Common Properties and Methods of Canvas (see page 2184)	The following table lists the commonly used properties of the Canvas object. ***Common properties of the Canvas object***
Copying Graphics to the Clipboard (see page 2186)	You can copy any picture, including the contents of image controls, to the clipboard. Once on the clipboard, the picture is available to all applications. To copy a picture to the clipboard, assign the picture to the clipboard object using the *Assign* method. This code shows how to copy the picture from an image control named *Image* to the clipboard in response to a click on an **Edit ▶ Copy** menu item:
Cutting Graphics to the Clipboard (see page 2186)	Cutting a graphic to the clipboard is exactly like copying it, but you also erase the graphic from the source. To cut a graphic from a picture to the clipboard, first copy it to the clipboard, then erase the original. In most cases, the only issue with cutting is how to show that the original image is erased. Setting the area to white is a common solution, as shown in the following code that attaches an event handler to the *OnClick* event of the **Edit ▶ Cut** menu item:
Drawing Lines (see page 2187)	To draw a straight line on a canvas, use the LineTo method of the canvas. *LineTo* draws a line from the current pen position to the point you specify and makes the endpoint of the line the current position. The canvas draws the line using its pen. For example, the following method draws crossed diagonal lines across a form whenever the form is painted:
Drawing Lines and Polylines (see page 2187)	A canvas can draw straight lines and polylines. A straight line is just a line of pixels connecting two points. A polyline is a series of straight lines, connected end-to-end. The canvas draws all lines using its pen.
Drawing On a Graphic (see page 2187)	You don't need any components to manipulate your application's graphic objects. You can construct, draw on, save, and destroy graphic objects without ever drawing anything on screen. In fact, your applications rarely draw directly on a form. More often, an application operates on graphics and then uses an image control component to display the graphic on a form. Once you move the application's drawing to the graphic in the image control, it is easy to add printing, clipboard, and loading and saving operations for any graphic objects. graphic objects can be bitmap files, drawings, icons or whatever other graphics classes... more (see page 2187)
Drawing On the Bitmap (see page 2188)	To draw on a bitmap, use the image control's canvas and attach the mouse-event handlers to the appropriate events in the image control. Typically, you would use region operations (fills, rectangles, polylines, and so on). These are fast and efficient methods of drawing. An efficient way to draw images when you need to access individual pixels is to use the bitmap *ScanLine* property. For general-purpose usage, you can set up the bitmap pixel format to 24 bits and then treat the pointer returned from *ScanLine* as an array of RGB. Otherwise, you will need to know the native format of... more (see page 2188)
Drawing Polygons (see page 2189)	To draw a polygon with any number of sides on a canvas, call the *Polygon* method of the canvas. *Polygon* takes an array of points as its only parameter and connects the points with the pen, then connects the last point to the first to close the polygon. After drawing the lines, *Polygon* uses the brush to fill the area inside the polygon.
Drawing Polylines (see page 2189)	In addition to individual lines, the canvas can also draw polylines, which are groups of any number of connected line segments. To draw a polyline on a canvas, call the Polyline method of the canvas. The parameter passed to the *Polyline* method is an array of points. You can think of a polyline as performing a *MoveTo* on the first point and *LineTo* on each successive point. For drawing multiple lines, *Polyline* is faster than using the MoveTo method and the LineTo method because it eliminates a lot of call overhead.

Drawing Rectangles and Ellipses (see page 2189)	To draw a rectangle or ellipse on a canvas, call the canvas's Rectangle method or Ellipse method, passing the coordinates of a bounding rectangle. The *Rectangle* method draws the bounding rectangle; *Ellipse* draws an ellipse that touches all sides of the rectangle.
Drawing Rounded Rectangles (see page 2189)	To draw a rounded rectangle on a canvas, call the canvas's RoundRect method. The first four parameters passed to RoundRect are a bounding rectangle, just as for the Rectangle method or the Ellipse method. *RoundRect* takes two more parameters that indicate how to draw the rounded corners.
Drawing Shapes (see page 2190)	Canvases have methods for drawing different kinds of shapes. The canvas draws the outline of a shape with its pen, then fills the interior with its brush. The line that forms the border for the shape is controlled by the current *Pen* object. This topic describes: • Drawing rectangles and ellipses (see page 2189). • Drawing rounded rectangles. (see page 2189) • Drawing polygons (see page 2189).
Drawing Shapes (Code) (see page 2190)	Drawing shapes is just as easy as drawing lines. Each one takes a single statement; you just need the coordinates. Here's a rewrite of the *OnMouseUp* event handler that draws shapes for all four tools:
Getting the Pen Position (see page 2192)	The current drawing position—the position from which the pen begins drawing its next line—is called the pen position. The canvas stores its pen position in its PenPos property. Pen position affects the drawing of lines only; for shapes and text, you specify all the coordinates you need. To set the pen position, call the MoveTo method of the canvas. For example, the following code moves the pen position to the upper left corner of the canvas:
Handling Multiple Drawing Objects in Your Application (see page 2192)	Various drawing methods (rectangle, shape, line, and so on) are typically available on the toolbar and button panel. Applications can respond to clicks on speed buttons to set the desired drawing objects. This section describes how to: • Keep track of which drawing tool to use (see page 2193). • Change the tool with speed buttons (see page 2184). • Use drawing tools (see page 2206).
Keeping Track of Which Drawing Tool to Use (see page 2193)	A graphics program needs to keep track of what kind of drawing tool (such as a line, rectangle, ellipse, or rounded rectangle) a user might want to use at any given time. You could assign numbers to each kind of tool, but then you would have to remember what each number stands for. You can do that more easily by assigning mnemonic constant names to each number, but your code won"t be able to distinguish which numbers are in the proper range and of the right type. Fortunately, Delphi provides a means to handle both of these shortcomings. You can... more (see page 2193)
Loading a Picture from a File (see page 2194)	Your application should provide the ability to load a picture from a file if your application needs to modify the picture or if you want to store the picture outside the application so a person or another application can modify the picture. To load a graphics file into an image control, call the *LoadFromFile* method of the image control's *Picture* object. The following code gets a file name from an open picture file dialog box, and then loads that file into an image control named *Image*:
Loading and Saving Graphics Files (see page 2194)	Graphic images that exist only for the duration of one running of an application are of very limited value. Often, you either want to use the same picture every time, or you want to save a created picture for later use. The image component makes it easy to load pictures from a file and save them again. The components you use to load, save, and replace graphic images support many graphic formats including bitmap files, metafiles, glyphs, and so on. They also support installable graphic classes. The way to load and save graphics files is the similar to any other... more (see page 2194)
Making Scrollable Graphics (see page 2195)	The graphic need not be the same size as the form: it can be either smaller or larger. By adding a scroll box control to the form and placing the graphic image inside it, you can display graphics that are much larger than the form or even larger than the screen. To add a scrollable graphic first you add a *TScrollBox* component and then you add the image control.
Pasting Graphics from the Clipboard (see page 2195)	If the clipboard contains a bitmapped graphic, you can paste it into any image object, including image controls and the surface of a form.

Placing the Control (see page 2196)	You can place an image control anywhere on a form. If you take advantage of the image control's ability to size itself to its picture, you need to set the top left corner only. If the image control is a nonvisible holder for a bitmap, you can place it anywhere, just as you would a nonvisual component. If you drop the image control on a scroll box already aligned to the form's client area, this assures that the scroll box adds any scroll bars necessary to access offscreen portions of the image's picture. Then set the image control's properties.
Reading and Setting Pixels (see page 2196)	You will notice that every canvas has an indexed Pixels property that represents the individual colored points that make up the image on the canvas. You rarely need to access *Pixels* directly, it is available only for convenience to perform small actions such as finding or setting a pixel's color. **Note:** Setting and getting individual pixels is thousands of times slower than performing graphics operations on regions. Do not use the Pixel array property to access the image pixels of a general array. For high-performance access to image pixels, see the TBitmap.ScanLine property
Refining Line Drawing (see page 2196)	With fields in place to track various points, you can refine an application's line drawing.
Refreshing the Screen (see page 2196)	At certain times, the operating system determines that objects onscreen need to refresh their appearance, so it generates WM_PAINT messages on Windows, which the VCL routes to *OnPaint* events. If you have written an *OnPaint* event handler for that object, it is called when you use the *Refresh* method. The default name generated for the *OnPaint* event handler in a form is *FormPaint*. You may want to use the *Refresh* method at times to refresh a component or form. For example, you might call *Refresh* in the form's *OnResize* event handler to redisplay any graphics or if using the... more (see page 2196)
Replacing the Picture (see page 2197)	You can replace the picture in an image control at any time. If you assign a new graphic to a picture that already has a graphic, the new graphic replaces the existing one. To replace the picture in an image control, assign a new graphic to the image control's *Picture* object. Creating the new graphic is the same process you used to create the initial graphic , but you should also provide a way for the user to choose a size other than the default size used for the initial graphic. An easy way to provide that option is to... more (see page 2197)
Responding to a Mouse Move (see page 2198)	An *OnMouseMove* event occurs periodically when the user moves the mouse. The event goes to the object that was under the mouse pointer when the user pressed the button. This allows you to give the user some intermediate feedback by drawing temporary lines while the mouse moves. To respond to mouse movements, define an event handler for the *OnMouseMove* event. This example uses mouse-move events to draw intermediate shapes on a form while the user holds down the mouse button, thus providing some feedback to the user. The *OnMouseMove* event handler draws a line on a form to the location... more (see page 2198)
Responding to a Mouse-down Action (see page 2198)	Whenever the user presses a button on the mouse, an OnMouseDown event goes to the object the pointer is over. The object can then respond to the event. To respond to a mouse-down action, attach an event handler to the *OnMouseDown* event. The Code editor generates an empty handler for a mouse-down event on the form:
Responding to a Mouse-up Action (see page 2199)	An OnMouseUp event occurs whenever the user releases a mouse button. The event usually goes to the object the mouse cursor is over when the user presses the button, which is not necessarily the same object the cursor is over when the button is released. This enables you, for example, to draw a line as if it extended beyond the border of the form. To respond to mouse-up actions, define a handler for the *OnMouseUp* event.
Responding to the Mouse (see page 2199)	Your application can respond to the mouse actions: mouse-button down, mouse moved, and mouse-button up. It can also respond to a click (a complete press-and-release, all in one place) that can be generated by some kinds of keystrokes (such as pressing *Enter* in a modal dialog box). This topic describes: • What's in a mouse event (see page 2207). • Responding to a mouse-down action (see page 2198). • Responding to a mouse-up action (see page 2199). • Responding to a mouse move (see page 2198).

Rubber Banding Example (see page 2199)	This example describes the details of implementing the "rubber banding" effect in an graphics application that tracks mouse movements as the user draws a graphic at runtime. The example code covered in this topic is taken from a sample application located in the Demos\Doc\Graphexdirectory. The application draws lines and shapes on a window's canvas in response to clicks and drags: pressing a mouse button starts drawing, and releasing the button ends the drawing. To start with, the example code shows how to draw on the surface of the main form. Later examples demonstrate drawing on a bitmap. The following topics... more (see page 2199)
Saving a Picture to a File (see page 2200)	The picture object can load and save graphics in several formats, and you can create and register your own graphic-file formats so that picture objects can load and store them as well. To save the contents of an image control in a file, call the *SaveToFile* method of the image control's *Picture* object. The *SaveToFile* method requires the name of a file in which to save. If the picture is newly created, it might not have a file name, or a user might want to save an existing picture in a different file. In either case, the application needs to... more (see page 2200)
Setting the Brush Bitmap Property (see page 2201)	A brush's *Bitmap* property lets you specify a bitmap image for the brush to use as a pattern for filling shapes and other areas. The following example loads a bitmap from a file and assigns it to the Brush of the Canvas of Form1:
Setting the Initial Bitmap Size (see page 2201)	When you place an image control, it is simply a container. However, you can set the image control's Picture property at design time to contain a static graphic. The control can also load its picture from a file at runtime, as described in Loading And Saving Graphics Files (see page 2194).
Sharing Code Among Event Handlers (see page 2202)	Any time you find that many your event handlers use the same code, you can make your application more efficient by moving the repeated code into a routine that all event handlers can share.
Tracking Movement (see page 2202)	The problem with this example as the *OnMouseMove* event handler is currently written is that it draws the line to the current mouse position from the last *mouse position*, not from the original position. You can correct this by moving the drawing position to the origin point, then drawing to the current point:
Tracking the Origin Point (see page 2204)	When drawing lines, track the point where the line starts with the *Origin* field. *Origin* must be set to the point where the mouse-down event occurs, so the mouse-up event handler can use *Origin* to place the beginning of the line, as in this code:
Types of Graphic Objects (see page 2204)	The component library provides the following graphic objects. These objects have methods to draw on the canvas, which are described in Using Canvas methods to draw graphic objects (see page 2205) and to load and save to graphics files, as described in Loading and saving graphics files (see page 2194) ***Graphic object types***
Using Brushes (see page 2205)	The Brush property of a canvas controls the way you fill areas, including the interior of shapes. Filling an area with a brush is a way of changing a large number of adjacent pixels in a specified way. The brush has three properties you can manipulate: • Color property changes the fill color. • Style property changes the brush style. • Bitmap property uses a bitmap as a brush pattern. The values of these properties determine the way the canvas fills shapes or other areas. By default, every brush starts out white, with a solid style and no pattern bitmap. You can use... more (see page 2205)
Using Canvas Methods to Draw Graphic Objects (see page 2205)	This topic shows how to use some common methods to draw graphic objects. It covers: • Drawing lines and polylines. (see page 2187) • Drawing shapes. (see page 2190) • Drawing rounded rectangles. (see page 2189) • Drawing polygons. (see page 2189)

Using Drawing Tools (see page 2206)	Now that you can tell what tool to use, you must indicate how to draw the different shapes. The only methods that perform any drawing are the mouse-move and mouse-up handlers, and the only drawing code draws lines, no matter what tool is selected. To use different drawing tools, your code needs to specify how to draw, based on the selected tool. You add this instruction to each tool's event handler. This topic describes: • Drawing shapes (see page 2190). • Sharing code among event handlers (see page 2202).
Using Pens (see page 2206)	The Pen property of a canvas controls the way lines appear, including lines drawn as the outlines of shapes. Drawing a straight line is really just changing a group of pixels that lie between two points. The pen itself has four properties you can change: • Color property changes the pen color. • Width property changes the pen width. • Style property changes the pen style. • Mode property changes the pen mode. The values of these properties determine how the pen changes the pixels in the line. By default, every pen starts out black, with a width of 1 pixel, a solid style,... more (see page 2206)
Using the Clipboard with Graphics (see page 2206)	You can use the Windows clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. The VCL's clipboard object makes it easy to handle different kinds of information, including graphics. Before you can use the clipboard object in your application, you must add the Clipbrd unit to the uses clause of any unit that needs to access clipboard data.
Using the Properties of the Canvas Object (see page 2207)	With the Canvas object, you can set the properties of a pen for drawing lines, a brush for filling shapes, a font for writing text, and an array of pixels to represent the image. This topic describes: • Using pens (see page 2206). • Using brushes (see page 2205). • Reading and setting pixels (see page 2196).
What's in a Mouse Event (see page 2207)	A mouse event occurs when a user moves the mouse in the user interface of an application. The VCL has three mouse events. ***Mouse events***
Working with Graphics and Multimedia: Overview (see page 2208)	Graphics and multimedia elements can add polish to your applications. You can introduce these features into your application in a variety of ways. To add graphical elements, you can insert pre-drawn pictures at design time, create them using graphical controls at design time, or draw them dynamically at runtime. To add multimedia capabilities, you can use special components that can play audio and video clips. This following topics describe how to enhance your applications by introducing graphics or multimedia elements: • Overview of Graphics Programming (see page 2176) • Working with multimedia (see page 2177)

3.2.4.13.1 Changing the Brush Style

A brush style determines what pattern the canvas uses to fill shapes. It lets you specify various ways to combine the brush's color with any colors already on the canvas. The predefined styles include solid color, no color, and various line and hatch patterns.

To change the style of a brush, set its Style property to one of the predefined values: bsBDiagonal, bsClear, bsCross, bsDiagCross, bsFDiagonal, bsHorizontal, bsSolid, or bsVertical. Cross-platform applications include the predefined values of

bsDense1 through bsDense7.

This example sets brush styles by sharing a click-event handler for a set of eight brush-style buttons. All eight buttons are selected, the **Object Inspector** ▸ **Events** ▸ **OnClick** is set, and the OnClick handler is named SetBrushStyle.

Here is the handler code:

```
procedure TForm1.SetBrushStyle(Sender: TObject);
begin
  with Canvas.Brush do
  begin
    if Sender = SolidBrush then Style := bsSolid
   else if Sender = ClearBrush then Style := bsClear
   else if Sender = HorizontalBrush then Style := bsHorizontal
   else if Sender = VerticalBrush then Style := bsVertical
   else if Sender = FDiagonalBrush then Style := bsFDiagonal
   else if Sender = BDiagonalBrush then Style := bsBDiagonal
   else if Sender = CrossBrush then Style := bsCross
   else if Sender = DiagCrossBrush then Style := bsDiagCross;
  end;
end;
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
if (Sender == SolidBrush)
  Canvas->Brush->Style = bsSolid;
else if (Sender == ClearBrush)
  Canvas->Brush->Style = bsClear;
else if (Sender == HorizontalBrush)
  Canvas->Brush->Style = bsHorizontal;
else if (Sender == VerticalBrush)
  Canvas->Brush->Style = bsVertical;
else if (Sender == FDiagonalBrush)
  Canvas->Brush->Style = bsFDiagonal;
else if (Sender == BDiagonalBrush)
  Canvas->Brush->Style = bsBDiagonal;
else if (Sender == CrossBrush)
  Canvas->Brush->Style = bsCross;
else if (Sender == DiagCrossBrush)
  Canvas->Brush->Style = bsDiagCross;
}
void __fastcall TForm1::SetBrushStyle(TObject *Sender)
{
   if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Brush->Style = (TBrushStyle) ((TSpeedButton *)Sender)->Tag;
}
```

See Also

Changing the Brush Color (see page 2181)

Setting the Brush Bitmap Property (see page 2201)

3.2.4.13.2 Overview of Graphics Programming

In VCL applications, the graphics components defined in the Graphics unit encapsulate the Windows Graphics Device Interface (GDI), making it easy to add graphics to your Windows applications.

To draw graphics in an application, you draw on an object's *canvas*, rather than directly on the object. The canvas is a property of the object, and is itself an object. A main advantage of the canvas object is that it handles resources effectively and it manages the device context for you, so your programs can use the same methods regardless of whether you are drawing on the screen, to a printer, or on bitmaps or metafiles. Canvases are available only at runtime, so you do all your work with canvases by writing code.

Note: Since *TCanvas* is a wrapper resource manager around the Windows device context, you can also use all Windows GDI

functions on the canvas. The *Handle* property of the canvas is the device context Handle.

How graphic images appear in your application depends on the type of object whose canvas you draw on. If you are drawing directly onto the canvas of a control, the picture is displayed immediately. However, if you draw on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from the bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the image appears only when the control has an opportunity to process its *OnPaint* message (VCL applications).

When working with graphics, you often encounter the terms *drawing* and *painting*:

- Drawing is the creation of a single, specific graphic element, such as a line or a shape, with code. In your code, you tell an object to draw a specific graphic in a specific place on its canvas by calling a drawing method of the canvas.
- Painting is the creation of the entire appearance of an object. Painting usually involves drawing. That is, in response to *OnPaint* events, an object generally draws some graphics. An edit box, for example, paints itself by drawing a rectangle and then drawing some text inside. A shape control, on the other hand, paints itself by drawing a single graphic.

The following topics describe how to use graphics components to simplify your coding.

- Refreshing the screen (see page 2196)
- Types of graphic objects (see page 2204)
- Common properties and methods of canvases (see page 2184)
- Handling multiple drawing objects in an application (see page 2192)
- Drawing on a bitmap (see page 2187)
- Loading and saving graphics files (see page 2194)
- Using the Clipboard with Graphics (see page 2206)
- Rubber banding example (see page 2199)

See Also

Working with Multimedia (see page 2177)

3.2.4.13.3 Working with Multimedia

You can add multimedia components to your applications. To do this, you can use either the TAnimate component on the Win32 page or the TMediaPlayer component on the System category of the **Tool palette**. Use the animate component when you want to add silent video clips to your application. Use the media player component when you want to add audio and/or video clips to an application.

This topic discusses:

- Adding silent video clips to an application (see page 2179)
- Adding audio and/or video clips to an application (see page 2177)

See Also

Overview of Graphics Programming (see page 2176)

3.2.4.13.4 Adding Audio and/or Video Clips to an Application

With the media player component, you can add audio and/or video clips to your application. It opens a media device and plays, stops, pauses, records, etc., the audio and/or video clips used by the media device. The media device may be hardware or software.

Note: Audio support is not available in cross-platform applications.

To add an audio and/or video clip to an application:

1. Double-click the media player icon on the System category of the **Tool palette**. This automatically put a media player control on the form window in which you want the media feature.
2. Using the **Object Inspector**, select the Name*property* and enter a new name for your media player control. You will use this when you call the media player control. (Follow the standard rules for naming Delphi identifiers.) Always work directly with the **Object Inspector** when setting design time properties and creating event handlers.
3. Select the DeviceType property and choose the appropriate device type to open using the *AutoOpen* property or the *Open* method. (If *DeviceType* is dtAutoSelect the device type is selected based on the file extension of the media file specified by the *FileName* property.) For more information on device types and their functions, see the table below.
4. If the device stores its media in a file, specify the name of the media file using the *FileName* property. Select the FileName property, click the ellipsis (...) button, and choose a media file from any available local or network directories and click Open in the Open dialog. Otherwise, insert the hardware the media is stored in (disk, cassette, and so on) for the selected media device, at runtime.
5. Set the AutoOpen property to *True*. This way the media player automatically opens the specified device when the form containing the media player control is created at runtime. If *AutoOpen* is *False*, the device must be opened with a call to the Open method.
6. Set the AutoEnable property to *True* to automatically enable or disable the media player buttons as required at runtime; or, double-click the EnabledButtons property to set each button to *True* or *False* depending on which ones you want to enable or disable. The multimedia device is played, paused, stopped, and so on when the user clicks the corresponding button on the media player component. The device can also be controlled by the methods that correspond to the buttons (Play, Pause, Stop, Next, Previous, and so on).
7. Position the media player control bar on the form by either clicking and dragging it to the appropriate place on the form or by selecting the Align property and choosing the appropriate align position from the drop down list. If you want the media player to be invisible at runtime, set the Visible property to *False* and control the device by calling the appropriate methods (Play, Pause, Stop, Next, Previous, Step, Back, StartRecording, Eject).
8. Make any other changes to the TMediaPlayer control settings. For example, if the media requires a display window, set the Display property to the control that displays the media. If the device uses multiple tracks, set the Tracks property to the desired track.

Multimedia device types and their functions

Device Type	Software/Hardware used	Plays	Uses Tracks	Uses a Display Window
dtAVIVideo	AVI Video Player for Windows	AVI Video files	No	Yes
dtCDAudio	CD Audio Player for Windows or a CD Audio Player	CD Audio Disks	Yes	No
dtDAT	Digital Audio Tape Player	Digital Audio Tapes	Yes	No
dtDigitalVideo	Digital Video Player for Windows	AVI, MPG, MOV files	No	Yes
dtMMMovie	MM Movie Player	MM film	No	Yes
dtOverlay	Overlay device	Analog Video	No	Yes
dtScanner	Image Scanner	N/A for Play (scans images on Record)	No	No
dtSequencer	MIDI Sequencer for Windows	MIDI files	Yes	No
dtVCR	Video Cassette Recorder	Video Cassettes	No	Yes
dtWaveAudio	Wave Audio Player for Windows	WAV files	No	No

For more information on using the media player control, see the topic called Example of Adding Audio and/or Video Clips (see page 2179).

See Also

Example of Adding Audio and/or Video Clips (see page 2179)

Adding Silent Video Clips to an Application (see page 2179)

3.2.4.13.5 Adding Silent Video Clips to an Application

With the animation control, you can add silent video clips to your application:

To add silent videop clips

1. Double-click the *TAnimate* icon on the Win32 category of the **Tool palette**. This automatically puts an animation control on the form window in which you want to display the video clip.
2. Using the **Object Inspector**, select the Name property and enter a new *nam*e for your animation control. You will use this name when you call the animation control. (Follow the standard rules for naming Delphi identifiers). Always work directly with the **Object Inspector** when setting design time properties and creating event handlers.
3. Do one of the following:

- Select the CommonAVI property and choose one of the AVIs available from the drop-down list; or
- Select the resource of an AVI using the ResName or ResID properties. Use ResHandle to indicate the module that contains the resource identified by *ResName* or *ResID*; or
- Select the FileName property and click the ellipsis (...) button, choose an AVI file from any available local or network directories and click Open in the Open AVI or Open GIF dialog (Windows and cross-platform applications). This loads the AVI or GIF file into memory. If you want to display the first frame of the AVI or GIF clip on-screen until it is played using the *Active* property or the *Play* method, then set the Open property to *True*.

4. Set the Repetitions property to the number of times you want to the AVI or GIF clip to play. If this value is 0, then the sequence is repeated until the Stop method is called.
5. Make any other changes to the animation control settings. For example, *i*f you want to change the first frame displayed when animation control opens, then set the StartFrameproperty to the desired frame value.
6. Set the Active property to *True* using the drop-down list or write an event handler to run the AVI or GIF clip when a specific event takes place at runtime. For example, to activate the AVI or GIF clip when a button object is clicked, write the button's OnClick event specifying that. You may also call the Play method to specify when to play the AVI (VCL only).

   Note: If you make any changes to the form or any of the components on the form after setting *Active* to *True*, the *Active* property becomes *False* and you have to reset it to *True*. Do this either just before runtime or at runtime.

   For more information on using the animation control, see the topic called Example of adding silent video clips (see page 2180).

See Also

Example of Adding Silent Video Clips (see page 2180)

Adding Audio and/or Video Clips to an Application (see page 2177)

3.2.4.13.6 Example of Adding Audio and/or Video Clips (VCL Only)

This example runs an AVI video clip of a multimedia advertisement.

To run this example, create a new project and save the Unit1.pas file to FrmAd.pas and save the Project1.dpr file to DelphiAd.dpr. Then:

1. Double-click the media player icon on the System category of the **Tool palette**.
2. Using the **Object Inspector**, set the Name property of the media player to *VideoPlayer1*.
3. Select its DeviceType property and choose dtAVIVideo from the drop-down list.

4. Select its FileName property, click the ellipsis (...) button, locate and choose an AVI file. Click Open in the Open dialog.
5. Set its AutoOpen property to *True* and its Visible property to *False*.
6. Double-click the Animate icon from the Win32 category of the **Tool palette**. Set its AutoSize property to *False*, its Height property to 175 and Width property to 200. Click and drag the animation control to the top left corner of the form.
7. Click the media player to bring back focus to it. Select its Display property and choose Animate1 from the drop down list.
8. Click the form to bring focus to it and select its Name property and enter *Delphi_Ad*. Now resize the form to the size of the animation control.
9. Double-click the form's *OnActivate* event and write the following code to run the AVI video when the form is in focus:

```
VideoPlayer1.Play;
VideoPlayer1->Play();
```

Choose **Run** ▶ **Run** to execute the AVI video.

See Also

Adding Audio and/or Video Clips to an Application (see page 2177)

3.2.4.13.7 Example of Adding Silent Video Clips

Suppose you want to display an animated logo as the first screen that appears when your application starts. After the logo finishes playing the screen disappears.

To run this example, create a new project and save the Unit1.pas file as Frmlogo.pas and save the Project1.dpr file as Logo.dpr. Then:

1. Double-click the animate icon from the Win32 category of the **Tool palette**.
2. Using the **Object Inspector**, set its Name property to *Logo1*.
3. Select its FileName property, click the ellipsis (...) button, locate and choose an AVI file. Then click Open in the Open AVI dialog. This loads the AVI file into memory.
4. Position the animation control box on the form by clicking and dragging it to the top right hand side of the form.
5. Set its Repetitions property to 5.
6. Click the form to bring focus to it and set its Name property to *LogoForm1* and its Caption property to *Logo Window*. Now decrease the height of the form to right-center the animation control on it.
7. Double-click the form's *OnActivate* event and write the following code to run the AVI clip when the form is in focus at runtime:

```
Logo1.Active := True;
Logo1->Active = true;
```

8. Double-click the Label icon on the Standard category of the **Tool palette**. Select its Caption property and enter *Welcome to Cool Images 4.0*. Now select its Font property, click the ellipsis (...) button and choose Font Style: Bold, Size: 18, Color: Navy from the Font dialog and click OK. Click and drag the label control to center it on the form.
9. Click the animation control to bring focus back to it. Double-click its *OnStop* event and write the following code to close the form when the AVI file stops:

```
LogoForm1.Close;
LogoForm1->Close();
```

Select **Run** ▶ **Run** to execute the animated logo window.

See Also

Adding Silent Video Clips to an Application (see page 2179)

3.2.4.13.8 Adding a Field to a Form Object to Track Mouse Actions

To track whether a mouse button was pressed, you must add an object field to the form object. When you add a component to a form, Delphi adds a field that represents that component to the form object, so that you can refer to the component by the name of its field. You can also add your own fields to forms by editing the type declaration in the form unit's header file.

In the following example, the form needs to track whether the user has pressed a mouse button. To do that, it adds a Boolean field and sets its value when the user presses the mouse button.

To add a field to an object, edit the object's type definition, specifying the field identifier and type after the **public** directive at the bottom of the declaration.

Delphi "owns" any declarations before the **public** directive: that's where it puts the fields that represent controls and the methods that respond to events.

See Also

Responding to the Mouse (see page 2199)

Refining Line Drawing (see page 2196)

3.2.4.13.9 Adding an Image Control

An image control is a container component that allows you to display your bitmap objects. You use an image control to hold a bitmap that is not necessarily displayed all the time, or which an application needs to use to generate other pictures.

Note: Adding Graphics to Controls (see page 2154) shows how to use graphics in controls.

See Also

Placing the Control (see page 2196)

Making Scrollable Bitmaps (see page 2195)

Setting the Initial Bitmap Size (see page 2201)

Drawing On the Bitmap (see page 2188)

3.2.4.13.10 Changing the Brush Color

A brush's color determines what color the canvas uses to fill shapes. To change the fill color, assign a value to the brush's *Color* property. Brush is used for background color in text and line drawing so you typically set the background color property.

You can set the brush color just as you do the pen color, in response to a click on a color grid on the brush's toolbar :

```
procedure TForm1.BrushColorClick(Sender: TObject);
begin
Canvas.Brush.Color := BrushColor.ForegroundColor;
end;
void __fastcall TForm1::BrushColorClick(TObject *Sender)
{
  Canvas->Brush->Color = BrushColor->BackgroundColor;
}
```

See Also

Changing the Brush Style (see page 2175)

Setting the Brush Bitmap Property (see page 2201)

3.2.4.13.11 Changing the Pen Color

You can set the color of a pen as you would any other *Color* property at runtime. A pen's color determines the color of the lines the pen draws, including lines drawn as the boundaries of shapes, as well as other lines and polylines. To change the pen color, assign a value to the *Color* property of the pen.

To let the user choose a new color for the pen, put a color grid on the pen's toolbar. A color grid can set both foreground and background colors. For a non-grid pen style, you must consider the background color, which is drawn in the gaps between line segments. Background color comes from the Brush color property.

Since the user chooses a new color by clicking the grid, this code changes the pen's color in response to the *OnClick* event:

```
procedure TForm1.PenColorClick(Sender: TObject);
begin
Canvas.Pen.Color := PenColor.ForegroundColor;
end;
void __fastcall TForm1::PenColorClick(TObject *Sender)
{
  Canvas->Pen->Color = PenColor->ForegroundColor;
}
```

See Also

Changing the Pen Width (see page 2183)

Changing the Pen Style (see page 2182)

Changing the Pen Mode (see page 2182)

Getting the Pen Position (see page 2192)

3.2.4.13.12 Changing the Pen Mode

A pen's *Mode* property lets you specify various ways to combine the pen's color with the color on the canvas. For example, the pen could always be black, be an inverse of the canvas background color, inverse of the pen color, and so on.

See Also

Changing the Pen Color (see page 2182)

Changing the Pen Width (see page 2183)

Changing the Pen Style (see page 2182)

Getting the Pen Position (see page 2192)

3.2.4.13.13 Changing the Pen Style

A pen's *Style* property allows you to set solid lines, dashed lines, dotted lines, and so on.

The task of setting the properties of pen is an ideal case for having different controls share same event handler to handle events. To determine which control actually got the event, you check the *Sender* parameter.

To create one click-event handler for six pen-style buttons on a pen's toolbar, do the following:

1. Select all six pen-style buttons and select the **Object Inspector** ▶ **Events** ▶ **OnClick event** and in the Handler column, type `SetPenStyle`. The Code editor generates an empty click-event handler called *SetPenStyle* and attaches it to the *OnClick* events of all six buttons.
2. Fill in the click-event handler by setting the pen's style depending on the value of Sender, which is the control that sent the click event:

```
procedure TForm1.SetPenStyle(Sender: TObject);
begin
with Canvas.Pen do
begin
if Sender = SolidPen then Style := psSolid
else if Sender = DashPen then Style := psDash
else if Sender = DotPen then Style := psDot
else if Sender = DashDotPen then Style := psDashDot
else if Sender = DashDotDotPen then Style := psDashDotDot
else if Sender = ClearPen then Style := psClear;
end;
end;
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender == SolidPen)
    Canvas->Pen->Style = psSolid;
  else if (Sender == DashPen)
    Canvas->Pen->Style = psDash;
  else if (Sender == DotPen)
    Canvas->Pen->Style = psDot;
  else if (Sender == DashDotPen)
    Canvas->Pen->Style = psDashDot;
  else if (Sender == DashDotDotPen)
    Canvas->Pen->Style = psDashDotDot;
'  else if (Sender == ClearPen)
    Canvas->Pen->Style = psClear;
}
void __fastcall TForm1::SetPenStyle(TObject *Sender)
{
  if (Sender->InheritsFrom (__classid(TSpeedButton))
    Canvas->Pen->Style = (TPenStyle) ((TSpeedButton *)Sender)->Tag;
}
```

See Also

Changing the Pen Color (see page 2182)

Changing the Pen Width (see page 2183)

Changing the Pen Mode (see page 2182)

Getting the Pen Position (see page 2192)

3.2.4.13.14 Changing the Pen Width

A pen's width determines the thickness, in pixels, of the lines it draws.

Note: When the thickness is greater than 1, Windows always draws solid lines, regardless of the value of the pen's *Style* property.

To change the pen width, assign a numeric value to the pen's Width property.

Suppose you have a scroll bar on the pen's toolbar to set width values for the pen. And suppose you want to update the label next to the scroll bar to provide feedback to the user. Using the scroll bar's position to determine the pen width, you update the pen width every time the position changes.

This is how to handle the scroll bar's *OnChange* event:

```
procedure TForm1.PenWidthChange(Sender: TObject);
begin
Canvas.Pen.Width := PenWidth.Position;{ set the pen width directly }
PenSize.Caption := IntToStr(PenWidth.Position);{ convert to string for caption }
end;
void __fastcall TForm1::PenWidthChange(TObject *Sender)
{
```

```
  Canvas->Pen->Width = PenWidth->Position;         // set the pen width directly
  PenSize->Caption = IntToStr(PenWidth->Position); // convert to string
}
```

See Also

Changing the Pen Color (see page 2182)

Changing the Pen Style (see page 2182)

Changing the Pen Mode (see page 2182)

Getting the Pen Position (see page 2192)

3.2.4.13.15 Changing the Tool with Speed Buttons

Each drawing tool needs an associated *OnClick* event handler. Suppose your application had a toolbar button for each of four drawing tools: line, rectangle, ellipse, and rounded rectangle. You would attach the following event handlers to the *OnClick* events of the four drawing-tool buttons, setting *DrawingTool* to the appropriate value for each:

```
procedure TForm1.LineButtonClick(Sender: TObject);{ LineButton }
begin
DrawingTool := dtLine;
end;
procedure TForm1.RectangleButtonClick(Sender: TObject);{ RectangleButton }
begin
DrawingTool := dtRectangle;
end;
procedure TForm1.EllipseButtonClick(Sender: TObject);{ EllipseButton }
begin
DrawingTool := dtEllipse;
end;
procedure TForm1.RoundedRectButtonClick(Sender: TObject);{ RoundRectButton }
begin
DrawingTool := dtRoundRect;
end;
void __fastcall TForm1::LineButtonClick(TObject *Sender)  // LineButton
{
  DrawingTool = dtLine;
}
void __fastcall TForm1::RectangleButtonClick(TObject *Sender) // RectangleButton
{
  DrawingTool = dtRectangle;
}
void __fastcall TForm1::EllipseButtonClick(TObject *Sender)  // EllipseButton
{
  DrawingTool = dtEllipse;
}
void __fastcall TForm1::RoundedRectButtonClick(TObject *Sender) // RoundRectBtn
{
  DrawingTool = dtRoundRect;
}
```

See Also

Keeping Track of Which Drawing Tool to Use (see page 2193)

Using Drawing Tools (see page 2206)

3.2.4.13.16 Common Properties and Methods of Canvas

The following table lists the commonly used properties of the Canvas object.

Common properties of the Canvas object

Properties	Descriptions
Font	Specifies the font to use when writing text on the image. Set the properties of the TFont object to specify the font face, color, size, and style of the font.
Brush	Determines the color and pattern the canvas uses for filling graphical shapes and backgrounds. Set the properties of the TBrush object to specify the color and pattern or bitmap to use when filling in spaces on the canvas.
Pen	Specifies the kind of pen the canvas uses for drawing lines and outlining shapes. Set the properties of the TPen object to specify the color, style, width, and mode of the pen.
PenPos	Specifies the current drawing position of the pen.
Pixels	Specifies the color of the area of pixels within the current ClipRect.

These properties are described in more detail in Using the properties of the Canvas object (see page 2207).

Here is a list of several methods you can use:

Common methods of the Canvas object

Method	Descriptions
Arc	Draws an arc on the image along the perimeter of the ellipse bounded by the specified rectangle.
Chord	Draws a closed figure represented by the intersection of a line and an ellipse.
CopyRect	Copies part of an image from another canvas into the canvas.
Draw	Renders the graphic object specified by the Graphic parameter on the canvas at the location given by the coordinates (X, Y).
Ellipse	Draws the ellipse defined by a bounding rectangle on the canvas.
FillRect	Fills the specified rectangle on the canvas using the current brush.
FloodFill (VCL only)	Fills an area of the canvas using the current brush.
FrameRect (VCL only)	Draws a rectangle using the Brush of the canvas to draw the border.
LineTo	Draws a line on the canvas from PenPos to the point specified by X and Y, and sets the pen position to (X, Y).
MoveTo	Changes the current drawing position to the point (X,Y).
Pie	Draws a pie-shaped the section of the ellipse bounded by the rectangle (X1, Y1) and (X2, Y2) on the canvas.
Polygon	Draws a series of lines on the canvas connecting the points passed in and closing the shape by drawing a line from the last point to the first point.
Polyline	Draws a series of lines on the canvas with the current pen, connecting each of the points passed to it in Points.
Rectangle	Draws a rectangle on the canvas with its upper left corner at the point (X1, Y1) and its lower right corner at the point (X2, Y2). Use *Rectangle* to draw a box using Pen and fill it using Brush.
RoundRect	Draws a rectangle with rounded corners on the canvas.
StretchDraw	Draws a graphic on the canvas so that the image fits in the specified rectangle. The graphic image may need to change its magnitude or aspect ratio to fit.
TextHeight, TextWidth	Returns the height and width, respectively, of a string in the current font. Height includes leading between lines.
TextOut	Writes a string on the canvas, starting at the point (X,Y), and then updates the PenPos to the end of the string.
TextRect	Writes a string inside a region; any portions of the string that fall outside the region do not appear.

These methods are described in more detail in Using Canvas methods to draw graphic objects (see page 2205).

See Also

Refreshing the Screen (see page 2196)

Types of Graphic Objects (see page 2204)

3.2.4.13.17 Copying Graphics to the Clipboard

You can copy any picture, including the contents of image controls, to the clipboard. Once on the clipboard, the picture is available to all applications.

To copy a picture to the clipboard, assign the picture to the clipboard object using the *Assign* method.

This code shows how to copy the picture from an image control named *Image* to the clipboard in response to a click on an **Edit ▸ Copy** menu item:

```
procedure TForm1.Copy1Click(Sender: TObject);
begin
  Clipboard.Assign(Image.Picture)
end.
void __fastcall TForm1::Copy1Click(TObject *Sender)
{
  Clipboard()->Assign(Image->Picture);
}
```

See Also

Cutting Graphics to the Clipboard (see page 2186)

Pasting Graphics from the Clipboard (see page 2195)

3.2.4.13.18 Cutting Graphics to the Clipboard

Cutting a graphic to the clipboard is exactly like copying it, but you also erase the graphic from the source.

To cut a graphic from a picture to the clipboard, first copy it to the clipboard, then erase the original.

In most cases, the only issue with cutting is how to show that the original image is erased. Setting the area to white is a common solution, as shown in the following code that attaches an event handler to the *OnClick* event of the **Edit ▸ Cut** menu item:

```
procedure TForm1.Cut1Click(Sender: TObject);
var
ARect: TRect;
begin
Copy1Click(Sender);{ copy picture to clipboard }
with Image.Canvas do
begin
CopyMode := cmWhiteness;{ copy everything as white }
ARect := Rect(0, 0, Image.Width, Image.Height);{ get bitmap rectangle }
CopyRect(ARect, Image.Canvas, ARect);{ copy bitmap over itself }
CopyMode := cmSrcCopy;{ restore normal mode }
end;
end;
void __fastcall TForm1::Cut1Click(TObject *Sender)
{
  TRect ARect;
  Copy1Click(Sender);          // copy picture to clipboard
  Image->Canvas->CopyMode = cmWhiteness; // copy everything as white
  ARect = Rect(0, 0, Image->Width, Image->Height); // get dimensions of image
  Image->Canvas->CopyRect(ARect, Image->Canvas, ARect); // copy bitmap over self
  Image->Canvas->CopyMode = cmSrcCopy; // restore default mode
}
```

See Also

Copying Graphics to the Clipboard (see page 2186)

Pasting Graphics from the Clipboard (see page 2195)

3.2.4.13.19 Drawing Lines

To draw a straight line on a canvas, use the LineTo method of the canvas.

LineTo draws a line from the current pen position to the point you specify and makes the endpoint of the line the current position. The canvas draws the line using its pen.

For example, the following method draws crossed diagonal lines across a form whenever the form is painted:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
with Canvas do
begin
MoveTo(0, 0);
LineTo(ClientWidth, ClientHeight);
MoveTo(0, ClientHeight);
LineTo(ClientWidth, 0);
end;
end;
void __fastcall TForm1::FormPaint(TObject *Sender)
{
  Canvas->MoveTo(0,0);
  Canvas->LineTo(ClientWidth, ClientHeight);
  Canvas->MoveTo(0, ClientHeight);
  Canvas->LineTo(ClientWidth, 0);
}
```

See Also

Drawing Polylines (see page 2189)

3.2.4.13.20 Drawing Lines and Polylines

A canvas can draw straight lines and polylines. A straight line is just a line of pixels connecting two points. A polyline is a series of straight lines, connected end-to-end. The canvas draws all lines using its pen.

See Also

Drawing Shapes (see page 2190)

Drawing Polylines (see page 2189)

3.2.4.13.21 Drawing On a Graphic

You don't need any components to manipulate your application's graphic objects. You can construct, draw on, save, and destroy graphic objects without ever drawing anything on screen. In fact, your applications rarely draw directly on a form. More often, an application operates on graphics and then uses an image control component to display the graphic on a form.

Once you move the application's drawing to the graphic in the image control, it is easy to add printing, clipboard, and loading and saving operations for any graphic objects. graphic objects can be bitmap files, drawings, icons or whatever other graphics classes that have been installed such as jpeg graphics.

Note: Because you are drawing on an offscreen image such as a *TBitmap* canvas, the image is not displayed until a control copies from a bitmap onto the control's canvas. That is, when drawing bitmaps and assigning them to an image control, the

image appears only when the control has an opportunity to process its paint message. But if you are drawing directly onto the canvas property of a control, the picture object is displayed immediately.

See Also

Overview of Graphics Programming (see page 2176)

Using the Properties of the Canvas Object (see page 2207)

Using Canvas Methods to Draw Graphic Objects (see page 2205)

Handling Multiple Drawing Objects in Your Application (see page 2192)

Loading and Saving Graphics Files (see page 2194)

Using the Clipboard with Graphics (see page 2206)

Rubber Banding Example (see page 2199)

3.2.4.13.22 Drawing On the Bitmap

To draw on a bitmap, use the image control's canvas and attach the mouse-event handlers to the appropriate events in the image control. Typically, you would use region operations (fills, rectangles, polylines, and so on). These are fast and efficient methods of drawing.

An efficient way to draw images when you need to access individual pixels is to use the bitmap *ScanLine* property. For general-purpose usage, you can set up the bitmap pixel format to 24 bits and then treat the pointer returned from *ScanLine* as an array of RGB. Otherwise, you will need to know the native format of the *ScanLine* property. This example shows how to use *ScanLine* to get pixels one line at a time.

```
procedure TForm1.Button1Click(Sender: TObject);
// This example shows drawing directly to the Bitmap
var
x,y : integer;
Bitmap : TBitmap;
P : PByteArray;
begin
Bitmap := TBitmap.create;
try
Bitmap.LoadFromFile("C:\Program Files\Borland\Delphi 4\Images\Splash\256color\factory.bmp");
for y := 0 to Bitmap.height -1 do
begin
P := Bitmap.ScanLine[y];
for x := 0 to Bitmap.width -1 do
P[x] := y;
end;
canvas.draw(0,0,Bitmap);
finally
Bitmap.free;
end;
end;
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Graphics::TBitmap *pBitmap = new Graphics::TBitmap();
// This example shows drawing directly to the Bitmap
Byte *ptr;
try
{
pBitmap->LoadFromFile("C:\\Program
Files\\Borland\\CBuilder\\Images\\Splash\\256color\\factory.bmp ");
for (int y = 0; y < pBitmap->Height; y++)
{
ptr = pBitmap->ScanLine[y];
for (int x = 0; x < pBitmap->Width; x++)
```

```
ptr[x] = (Byte)y;
}
Canvas->Draw(0,0,pBitmap);
}
catch (...)
{
ShowMessage("Could not load or alter bitmap");
}
delete pBitmap;
}
```

See Also

Placing the Control (see page 2196)

Setting the Initial Bitmap Size (see page 2201)

3.2.4.13.23 Drawing Polygons

To draw a polygon with any number of sides on a canvas, call the *Polygon* method of the canvas.

Polygon takes an array of points as its only parameter and connects the points with the pen, then connects the last point to the first to close the polygon. After drawing the lines, *Polygon* uses the brush to fill the area inside the polygon.

See Also

Drawing Rectangles and Ellipses (see page 2189)

Drawing Rounded Rectangles (see page 2189)

3.2.4.13.24 Drawing Polylines

In addition to individual lines, the canvas can also draw polylines, which are groups of any number of connected line segments.

To draw a polyline on a canvas, call the Polyline method of the canvas.

The parameter passed to the *Polyline* method is an array of points. You can think of a polyline as performing a *MoveTo* on the first point and *LineTo* on each successive point. For drawing multiple lines, *Polyline* is faster than using the MoveTo method and the LineTo method because it eliminates a lot of call overhead.

See Also

Drawing Lines (see page 2187)

3.2.4.13.25 Drawing Rectangles and Ellipses

To draw a rectangle or ellipse on a canvas, call the canvas's Rectangle method or Ellipse method, passing the coordinates of a bounding rectangle.

The *Rectangle* method draws the bounding rectangle; *Ellipse* draws an ellipse that touches all sides of the rectangle.

See Also

Drawing Rounded Rectangles (see page 2189)

Drawing Polygons (see page 2189)

3.2.4.13.26 Drawing Rounded Rectangles

To draw a rounded rectangle on a canvas, call the canvas's RoundRect method.

The first four parameters passed to RoundRect are a bounding rectangle, just as for the Rectangle method or the Ellipse method. *RoundRect* takes two more parameters that indicate how to draw the rounded corners.

See Also

Drawing Rectangles and Ellipses (see page 2189)

Drawing Polygons (see page 2189)

3.2.4.13.27 Drawing Shapes

Canvases have methods for drawing different kinds of shapes. The canvas draws the outline of a shape with its pen, then fills the interior with its brush. The line that forms the border for the shape is controlled by the current *Pen* object.

This topic describes:

- Drawing rectangles and ellipses (see page 2189).
- Drawing rounded rectangles. (see page 2189)
- Drawing polygons (see page 2189).

See Also

Drawing Lines and Polylines (see page 2187)

Using Pens (see page 2206)

Using Brushes (see page 2205)

3.2.4.13.28 Drawing Shapes (Code)

Drawing shapes is just as easy as drawing lines. Each one takes a single statement; you just need the coordinates.

Here's a rewrite of the *OnMouseUp* event handler that draws shapes for all four tools:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button TMouseButton; Shift: TShiftState;
                             X,Y: Integer);
begin
case DrawingTool of
dtLine:
begin
Canvas.MoveTo(Origin.X, Origin.Y);
Canvas.LineTo(X, Y)
end;
dtRectangle: Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
dtEllipse: Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
dtRoundRect: Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
(Origin.X - X) div 2, (Origin.Y - Y) div 2);
end;
Drawing := False;
end;
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
                                    TShiftState Shift, int X, int Y){
  switch (DrawingTool)
  {
    case dtLine:
      Canvas->MoveTo(Origin.x, Origin.y);
      Canvas->LineTo(X, Y);
      break;
    case dtRectangle:
      Canvas->Rectangle(Origin.x, Origin.y, X, Y);
      break;
    case dtEllipse:
```

```
      Canvas->Ellipse(Origin.x, Origin.y, X, Y);
      break;
    case dtRoundRect:
      Canvas->Rectangle(Origin.x, Origin.y, X, Y, (Origin.x - X)/2,
                        (Origin.y - Y)/2);
      break;
  }
  Drawing = false;
}
```

Of course, you also need to update the *OnMouseMove* handler to draw shapes:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin
if Drawing then
begin
Canvas.Pen.Mode := pmNotXor;
case DrawingTool of
dtLine: begin
Canvas.MoveTo(Origin.X, Origin.Y);
Canvas.LineTo(MovePt.X, MovePt.Y);
Canvas.MoveTo(Origin.X, Origin.Y);
Canvas.LineTo(X, Y);
end;
dtRectangle: begin
Canvas.Rectangle(Origin.X, Origin.Y, MovePt.X, MovePt.Y);
Canvas.Rectangle(Origin.X, Origin.Y, X, Y);
end;
dtEllipse: begin
Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
Canvas.Ellipse(Origin.X, Origin.Y, X, Y);
end;
dtRoundRect: begin
Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
(Origin.X - X) div 2, (Origin.Y - Y) div 2);
Canvas.RoundRect(Origin.X, Origin.Y, X, Y,
(Origin.X - X) div 2, (Origin.Y - Y) div 2);
end;
end;
MovePt := Point(X, Y);
end;
Canvas.Pen.Mode := pmCopy;
end;
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->Pen->Mode = pmNotXor;        // use XOR mode to draw/erase
    switch (DrawingTool)
    {
      case dtLine:
        Canvas->MoveTo(Origin.x, Origin.y);
        Canvas->LineTo(MovePt.x, MovePt.y);
        Canvas->MoveTo(Origin.x, Origin.y);
        Canvas->LineTo(X, Y);
        break;
      case dtRectangle:
        Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y);
        Canvas->Rectangle(Origin.x, Origin.y, X, Y);
        break;
      case dtEllipse:
        Canvas->Ellipse(Origin.x, Origin.y, MovePt.x, MovePt.y);
        Canvas->Ellipse(Origin.x, Origin.y, X, Y);
        break;
      case dtRoundRect:
        Canvas->Rectangle(Origin.x, Origin.y, MovePt.x, MovePt.y,
```

```
                        (Origin.x - MovePt.x)/2,(Origin.y - MovePt.y)/2);
          Canvas->Rectangle(Origin.x, Origin.y, X, Y,
                        (Origin.x - X)/2, (Origin.y - Y)/2);
          break;
      }
      MovePt = Point(X, Y);
    }
    Canvas->Pen->Mode = pmCopy;
  }
```

Typically, all the repetitious code that is in the above example would be in a separate routine. The next topic shows all the shape-drawing code in a single routine that all mouse-event handlers can call.

See Also

Sharing Code Among Event Handlers (see page 2202)

3.2.4.13.29 Getting the Pen Position

The current drawing position—the position from which the pen begins drawing its next line—is called the pen position. The canvas stores its pen position in its PenPos property. Pen position affects the drawing of lines only; for shapes and text, you specify all the coordinates you need.

To set the pen position, call the MoveTo method of the canvas. For example, the following code moves the pen position to the upper left corner of the canvas:

```
Canvas.MoveTo(0, 0);
Canvas->MoveTo(0, 0);
```

Note: Drawing a line with the LineTo method also moves the current position to the endpoint of the line.

See Also

Changing the Pen Color (see page 2182)

Changing the Pen Width (see page 2183)

Changing the Pen Style (see page 2182)

Changing the Pen Mode (see page 2182)

3.2.4.13.30 Handling Multiple Drawing Objects in Your Application

Various drawing methods (rectangle, shape, line, and so on) are typically available on the toolbar and button panel. Applications can respond to clicks on speed buttons to set the desired drawing objects. This section describes how to:

- Keep track of which drawing tool to use (see page 2193).
- Change the tool with speed buttons (see page 2184).
- Use drawing tools (see page 2206).

See Also

Overview of Graphics Programming (see page 2176)

Using the Properties of the Canvas Object (see page 2207)

Using Canvas Methods to Draw Graphic Objects (see page 2205)

Drawing on a Bitmap (see page 2188)

Loading and Saving Graphics Files (see page 2194)

Using the Clipboard with Graphics (see page 2206)

Rubber Banding Example (see page 2199)

3.2.4.13.31 Keeping Track of Which Drawing Tool to Use

A graphics program needs to keep track of what kind of drawing tool (such as a line, rectangle, ellipse, or rounded rectangle) a user might want to use at any given time.

You could assign numbers to each kind of tool, but then you would have to remember what each number stands for. You can do that more easily by assigning mnemonic constant names to each number, but your code won"t be able to distinguish which numbers are in the proper range and of the right type. Fortunately, Delphi provides a means to handle both of these shortcomings. You can declare an enumerated type.

An enumerated type is really just a shorthand way of assigning sequential values to constants. Since it's also a type declaration, you can use Delphi's type-checking to ensure that you assign only those specific values.

To declare an enumerated type, use the reserved work type, followed by an identifier for the type, then an equal sign, and the identifiers for the values in the type in parentheses, separated by commas.

For example, the following code declares an enumerated type for each drawing tool available in a graphics application:

```
type
TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
typedef enum {dtLine, dtRectangle, dtEllipse, dtRoundRect} TDrawingTool;
```

By convention, type identifiers begin with the letter *T*, and groups of similar constants (such as those making up an enumerated type) begin with a 2-letter prefix (such as *dt* for "drawing tool").

The declaration of the TDrawingTool type is equivalent to declaring a group of constants:

```
const
  dtLine = 0;
  dtRectangle = 1;
  dtEllipse = 2;
  dtRoundRect = 3;
```

The main difference is that by declaring the enumerated type, you give the constants not just a value, but also a type, which enables you to use the Delphi language's type-checking to prevent many errors. A variable of type *TDrawingTool* can be assigned only one of the constants dtLine..dtRoundRect. Attempting to assign some other number (even one in the range 0..3) generates a compile-time error.

In the following code, a field added to a form keeps track of the form's drawing tool:

```
type
TDrawingTool = (dtLine, dtRectangle, dtEllipse, dtRoundRect);
TForm1 = class(TForm)
...{ method declarations }
public
Drawing: Boolean;
Origin, MovePt: TPoint;
DrawingTool: TDrawingTool;{ field to hold current tool }
end;
enum TDrawingTool {dtLine, dtRectangle, dtEllipse, dtRoundRect};
class TForm1 : public TForm
{
__published:  // IDE-managed Components
  void __fastcall FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
  void __fastcall FormMouseMove(TObject *Sender, TShiftState Shift, int X,
  int Y);
  void __fastcall FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y);
private:
public:
```

```
    __fastcall TForm1(TComponent* Owner);
    bool Drawing;  //field to track whether button was pressed
    TDrawingTool DrawingTool;  // field to hold current tool
};
```

See Also

Changing the Tool with Speed Buttons (see page 2184)

Using Drawing Tools (see page 2206)

3.2.4.13.32 Loading a Picture from a File

Your application should provide the ability to load a picture from a file if your application needs to modify the picture or if you want to store the picture outside the application so a person or another application can modify the picture.

To load a graphics file into an image control, call the *LoadFromFile* method of the image control's *Picture* object.

The following code gets a file name from an open picture file dialog box, and then loads that file into an image control named *Image*:

```
procedure TForm1.Open1Click(Sender: TObject);
begin
if OpenPictureDialog1.Execute then
begin
CurrentFile := OpenPictureDialog1.FileName;
Image.Picture.LoadFromFile(CurrentFile);
end;
end;
void __fastcall TForm1::Open1Click(TObject *Sender)
{
  if (OpenPictureDialog1->Execute())
  {
    CurrentFile = OpenPictureDialog1->FileName;
    Image->Picture->LoadFromFile(CurrentFile);
  }
}
```

See Also

Saving a Picture to a File (see page 2200)

Replacing the Picture (see page 2197)

3.2.4.13.33 Loading and Saving Graphics Files

Graphic images that exist only for the duration of one running of an application are of very limited value. Often, you either want to use the same picture every time, or you want to save a created picture for later use. The image component makes it easy to load pictures from a file and save them again.

The components you use to load, save, and replace graphic images support many graphic formats including bitmap files, metafiles, glyphs, and so on. They also support installable graphic classes.

The way to load and save graphics files is the similar to any other files and is described in these topics:

- Loading a picture from a file. (see page 2194)
- Saving a picture to a file. (see page 2200)
- Replacing the picture. (see page 2197)

See Also

Overview of Graphics Programming (see page 2176)

Using the Properties of the Canvas Object (see page 2207)

Using Canvas Methods to Draw Graphic Objects (see page 2205)

Handling Multiple Drawing Objects in Your Application (see page 2192)

Drawing on a Bitmap (see page 2188)

Using the Clipboard with Graphics (see page 2206)

Rubber Banding Example (see page 2199)

3.2.4.13.34 Making Scrollable Graphics

The graphic need not be the same size as the form: it can be either smaller or larger. By adding a scroll box control to the form and placing the graphic image inside it, you can display graphics that are much larger than the form or even larger than the screen. To add a scrollable graphic first you add a *TScrollBox* component and then you add the image control.

See Also

Adding an Image Control (see page 2181)

3.2.4.13.35 Pasting Graphics from the Clipboard

If the clipboard contains a bitmapped graphic, you can paste it into any image object, including image controls and the surface of a form.

To paste a graphic from the clipboard:

1. Call the clipboard's HasFormat method (VCL applications) to see whether the clipboard contains a graphic. HasFormat is a Boolean function. It returns *True* if the clipboard contains an item of the type specified in the parameter. To test for graphics on the Windows platform, you pass *CF_BITMAP*.
2. Assign the clipboard to the destination.

   Note: The following VCL code shows how to paste a picture from the clipboard into an image control in response to a click on an Edit->Paste

   menu item:

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
Bitmap: TBitmap;
begin
if Clipboard.HasFormat(CF_BITMAP) then { is there a bitmap on the Windows clipboard? )
begin
Image1.Picture.Bitmap.Assign(Clipboard);
 end;
end;
void __fastcall TForm1::Paste1Click(TObject *Sender)
{
Graphics::TBitmap *Bitmap;
if (Clipboard()->HasFormat(CF_BITMAP)){
Image1->Picture->Bitmap->Assign(Clipboard());
  }
}
```

The graphic on the clipboard could come from this application, or it could have been copied from another application, such as Microsoft Paint. You do not need to check the clipboard format in this case because the paste menu should be disabled when the clipboard does not contain a supported format.

See Also

Copying Graphics to the Clipboard (see page 2186)

Cutting Graphics to the Clipboard (see page 2186)

3.2.4.13.36 Placing the Control

You can place an image control anywhere on a form. If you take advantage of the image control's ability to size itself to its picture, you need to set the top left corner only. If the image control is a nonvisible holder for a bitmap, you can place it anywhere, just as you would a nonvisual component.

If you drop the image control on a scroll box already aligned to the form's client area, this assures that the scroll box adds any scroll bars necessary to access offscreen portions of the image's picture. Then set the image control's properties.

See Also

Setting the Initial Bitmap Size (see page 2201)

Drawing On the Bitmap (see page 2188)

3.2.4.13.37 Reading and Setting Pixels

You will notice that every canvas has an indexed Pixels property that represents the individual colored points that make up the image on the canvas. You rarely need to access *Pixels* directly, it is available only for convenience to perform small actions such as finding or setting a pixel's color.

Note: Setting and getting individual pixels is thousands of times slower than performing graphics operations on regions. Do not use the Pixel array property to access the image pixels of a general array. For high-performance access to image pixels, see the TBitmap.ScanLine property

See Also

Using Pens (see page 2206)

Using Brushes (see page 2205)

3.2.4.13.38 Refining Line Drawing

With fields in place to track various points, you can refine an application's line drawing.

See Also

Responding to the Mouse (see page 2199)

Adding a Field to a Form Object to Track Mouse Actions (see page 2181)

3.2.4.13.39 Refreshing the Screen

At certain times, the operating system determines that objects onscreen need to refresh their appearance, so it generates WM_PAINT messages on Windows, which the VCL routes to *OnPaint* events. If you have written an *OnPaint* event handler for that object, it is called when you use the *Refresh* method. The default name generated for the *OnPaint* event handler in a form is *FormPaint*. You may want to use the *Refresh* method at times to refresh a component or form. For example, you might call *Refresh* in the form's *OnResize* event handler to redisplay any graphics or if using the VCL, you want to paint a background on a form.

While some operating systems automatically handle the redrawing of the client area of a window that has been invalidated,

Windows does not. In the Windows operating system anything drawn on the screen is permanent. When a form or control is temporarily obscured, for example during window dragging, the form or control must repaint the obscured area when it is re-exposed. For more information about the WM_PAINT message, see the Windows online Help.

If you use the *TImage* control to display a graphical image on a form, the painting and refreshing of the graphic contained in the *TImage* is handled automatically. The *Picture* property specifies the actual bitmap, drawing, or other graphic object that *TImage* displays. You can also set the *Proportional* property to ensure that the image can be fully displayed in the image control without any distortion. Drawing on a *TImage* creates a persistent image. Consequently, you do not need to do anything to redraw the contained image. In contrast, *TPaintBox*'s canvas maps directly onto the screen device (VCL applications), so that anything drawn to the *PaintBox*'s canvas is transitory. This is true of nearly all controls, including the form itself. Therefore, if you draw or paint on a *TPaintBox* in its constructor, you will need to add that code to your *OnPaint* event handler in order for the image to be repainted each time the client area is invalidated.

See Also

Common Properties and Methods of Canvas (see page 2184)

Types of Graphic Objects (see page 2204)

3.2.4.13.40 Replacing the Picture

You can replace the picture in an image control at any time. If you assign a new graphic to a picture that already has a graphic, the new graphic replaces the existing one.

To replace the picture in an image control, assign a new graphic to the image control's *Picture* object.

Creating the new graphic is the same process you used to create the initial graphic , but you should also provide a way for the user to choose a size other than the default size used for the initial graphic. An easy way to provide that option is to present a dialog box.

With such a dialog box in your project, add it to the uses clause in the unit for your main form. You can then attach an event handler to the **File ▶ New** menu item's OnClick event. Here's an example:

```
procedure TForm1.New1Click(Sender: TObject);
var
Bitmap: TBitmap;{ temporary variable for the new bitmap }
begin
with NewBMPForm do
begin
ActiveControl := WidthEdit;{ make sure focus is on width field }
WidthEdit.Text := IntToStr(Image.Picture.Graphic.Width);{ use current dimensions... }
HeightEdit.Text := IntToStr(Image.Picture.Graphic.Height);{ ...as default }
if ShowModal <> idCancel then{ continue if user doesn"t cancel dialog box }
begin
Bitmap := TBitmap.Create;{ create fresh bitmap object }
Bitmap.Width := StrToInt(WidthEdit.Text);{ use specified width }
Bitmap.Height := StrToInt(HeightEdit.Text);{ use specified height }
Image.Picture.Graphic := Bitmap;{ replace graphic with new bitmap }
CurrentFile := "";{ indicate unnamed file }
     Bitmap.Free;
end;
end;
end;
void __fastcall TForm1::New1Click(TObject *Sender)
{
Graphics::TBitmap *Bitmap;
  // make sure focus is on width field
 NewBMPForm->ActiveControl = NewBMPForm->WidthEdit;
  // initialize to current dimensions as default ...
     NewBMPForm->WidthEdit->Text = IntToStr(Image->Picture->Graphic->Width);
NewBMPForm->HeightEdit->Text = IntToStr(Image->Picture->Graphic->Height);
```

```
  if (NewBMPForm->ShowModal() != IDCANCEL){        // if user does not cancel dialog...
Bitmap = new Graphics::TBitmap();             // create a new bitmap object
    // use specified dimensions
     Bitmap->Width = StrToInt(NewBMPForm->WidthEdit->Text);
Bitmap->Height = StrToInt(NewBMPForm->HeightEdit->Text);
Image->Picture->Graphic = Bitmap;            // replace graphic with new bitmap
CurrentFile = EmptyStr;                      //indicate unnamed file
        delete Bitmap;
}
}
```

Note: Assigning a new bitmap to the picture object's Graphic property causes the picture object to copy the new graphic, but it does not take ownership of it. The picture object maintains its own internal graphic object. Because of this, the previous code frees the bitmap object after making the assignment.

See Also

Loading a Picture from a File (see page 2194)

Saving a Picture to a File (see page 2200)

3.2.4.13.41 Responding to a Mouse Move

An *OnMouseMove* event occurs periodically when the user moves the mouse. The event goes to the object that was under the mouse pointer when the user pressed the button. This allows you to give the user some intermediate feedback by drawing temporary lines while the mouse moves.

To respond to mouse movements, define an event handler for the *OnMouseMove* event. This example uses mouse-move events to draw intermediate shapes on a form while the user holds down the mouse button, thus providing some feedback to the user. The *OnMouseMove* event handler draws a line on a form to the location of the *OnMouseMove* event:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
Canvas.LineTo(X, Y);{ draw line to current position }
end;
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->LineTo(X, Y);
}
```

With this code, moving the mouse over the form causes drawing to follow the mouse, even before the mouse button is pressed.

Mouse-move events occur even when you haven't pressed the mouse button.

If you want to track whether there is a mouse button pressed, you need to add an object field to the form object.

See Also

What's in a Mouse Event (see page 2207)

Responding to a Mouse-down Action (see page 2198)

Responding to a Mouse-up Action (see page 2199)

3.2.4.13.42 Responding to a Mouse-down Action

Whenever the user presses a button on the mouse, an OnMouseDown event goes to the object the pointer is over. The object can then respond to the event.

To respond to a mouse-down action, attach an event handler to the *OnMouseDown* event.

The Code editor generates an empty handler for a mouse-down event on the form:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
end;
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
}
```

See Also

What's in a Mouse Event (see page 2207)

Responding to a Mouse-up Action (see page 2199)

Responding to a Mouse Move (see page 2198)

3.2.4.13.43 Responding to a Mouse-up Action

An OnMouseUp event occurs whenever the user releases a mouse button. The event usually goes to the object the mouse cursor is over when the user presses the button, which is not necessarily the same object the cursor is over when the button is released. This enables you, for example, to draw a line as if it extended beyond the border of the form.

To respond to mouse-up actions, define a handler for the *OnMouseUp* event.

See Also

What's in a Mouse Event (see page 2207)

Responding to a Mouse-down Action (see page 2198)

Responding to a Mouse Move (see page 2198)

3.2.4.13.44 Responding to the Mouse

Your application can respond to the mouse actions: mouse-button down, mouse moved, and mouse-button up. It can also respond to a click (a complete press-and-release, all in one place) that can be generated by some kinds of keystrokes (such as pressing *Enter* in a modal dialog box).

This topic describes:

- What's in a mouse event (see page 2207).
- Responding to a mouse-down action (see page 2198).
- Responding to a mouse-up action (see page 2199).
- Responding to a mouse move (see page 2198).

See Also

Adding a Field to a Form Object to Track Mouse Actions (see page 2181)

Refining Line Drawing (see page 2196)

What's in a Mouse Event (see page 2207)

3.2.4.13.45 Rubber Banding Example

This example describes the details of implementing the "rubber banding" effect in an graphics application that tracks mouse movements as the user draws a graphic at runtime. The example code covered in this topic is taken from a sample application

located in the Demos\Doc\Graphexdirectory. The application draws lines and shapes on a window's canvas in response to clicks and drags: pressing a mouse button starts drawing, and releasing the button ends the drawing.

To start with, the example code shows how to draw on the surface of the main form. Later examples demonstrate drawing on a bitmap.

The following topics describe the example:

- Responding to the mouse (see page 2199).
- Adding a field to a form object to track mouse actions (see page 2181).
- Refining line drawing (see page 2196).

See Also

Overview of Graphics Programming (see page 2176)

Using the Properties of the Canvas Object (see page 2207)

Using Canvas Methods to Draw Graphic Objects (see page 2205)

Handling Multiple Drawing Objects in Your Application (see page 2192)

Drawing on a Bitmap (see page 2188)

Loading and Saving Graphics Files (see page 2194)

Using the Clipboard with Graphics (see page 2206)

3.2.4.13.46 Saving a Picture to a File

The picture object can load and save graphics in several formats, and you can create and register your own graphic-file formats so that picture objects can load and store them as well.

To save the contents of an image control in a file, call the *SaveToFile* method of the image control's *Picture* object.

The *SaveToFile* method requires the name of a file in which to save. If the picture is newly created, it might not have a file name, or a user might want to save an existing picture in a different file. In either case, the application needs to get a file name from the user before saving, as shown in the next topic.

The following pair of event handlers, attached to the **File▶Save** and **File▶Save As** menu items, respectively, handle the resaving of named files, saving of unnamed files, and saving existing files under new names.

```
procedure TForm1.Save1Click(Sender: TObject);
begin
if CurrentFile <> '' then
Image.Picture.SaveToFile(CurrentFile){ save if already named }
else SaveAs1Click(Sender);{ otherwise get a name }
end;
procedure TForm1.Saveas1Click(Sender: TObject);
begin
if SaveDialog1.Execute then{ get a file name }
begin
CurrentFile := SaveDialog1.FileName;{ save the user-specified name }
Save1Click(Sender);{ then save normally }
end;
end;
void __fastcall TForm1::Save1Click(TObject *Sender)
{
  if (!CurrentFile.IsEmpty())
    Image->Picture->SaveToFile(CurrentFile);   // save if already named
else SaveAs1Click(Sender);                     // otherwise get a name
}
void __fastcall TForm1::SaveAs1Click(TObject *Sender)
```

```
{
  if (SaveDialog1->Execute())              // get a file name
  {
    CurrentFile = SaveDialog1->FileName;  // save user-specified name
    Save1Click(Sender);                   // then save normally
  }
}
```

See Also

Loading a Picture from a File (see page 2194)

Replacing the Picture (see page 2197)

3.2.4.13.47 Setting the Brush Bitmap Property

A brush's *Bitmap* property lets you specify a bitmap image for the brush to use as a pattern for filling shapes and other areas.

The following example loads a bitmap from a file and assigns it to the Brush of the Canvas of Form1:

```
var
Bitmap: TBitmap;
begin
Bitmap := TBitmap.Create;
try
Bitmap.LoadFromFile('MyBitmap.bmp');
Form1.Canvas.Brush.Bitmap := Bitmap;
Form1.Canvas.FillRect(Rect(0,0,100,100));
finally
Form1.Canvas.Brush.Bitmap := nil;
Bitmap.Free;
end;
end;
BrushBmp->LoadFromFile("MyBitmap.bmp");
Form1->Canvas->Brush->Bitmap = BrushBmp;
Form1->Canvas->FillRect(Rect(0,0,100,100));
```

Note: The brush does not assume ownership of a bitmap object assigned to its *Bitmap* property. You must ensure that the Bitmap object remains valid for the lifetime of the Brush, and you must free the Bitmap object yourself afterwards.

See Also

Changing the Brush Color (see page 2181)

Changing the Brush Style (see page 2175)

3.2.4.13.48 Setting the Initial Bitmap Size

When you place an image control, it is simply a container. However, you can set the image control's Picture property at design time to contain a static graphic. The control can also load its picture from a file at runtime, as described in Loading And Saving Graphics Files (see page 2194).

To create a blank bitmap when the application starts

1. Attach a handler to the *OnCreate* event for the form that contains the image.
2. Create a bitmap object, and assign it to the image control's *Picture.Graphic* property.

In this example, the image is in the application's main form, *Form1*, so the code attaches a handler to *Form1's OnCreate* event:

```
procedure TForm1.FormCreate(Sender: TObject);
var
Bitmap: TBitmap;{ temporary variable to hold the bitmap }
begin
```

```
Bitmap := TBitmap.Create;{ construct the bitmap object }
  Bitmap.Width := 200;{ assign the initial width... }
  Bitmap.Height := 200;{ ...and the initial height }
  Image.Picture.Graphic := Bitmap;{ assign the bitmap to the image control }
  Bitmap.Free; {We are done with the bitmap, so free it }
end;
void __fastcall TForm1::FormCreate(TObject *Sender)
{
  Graphics::TBitmap *Bitmap = new Graphics::TBitmap();   // create the bitmap object
  Bitmap->Width = 200;               // assign the initial width...
  Bitmap->Height = 200;              // ...and the initial height
  Image->Picture->Graphic = Bitmap;  // assign the bitmap to the image control
  delete Bitmap;                     // free the bitmap object
}
```

Assigning the bitmap to the picture's *Graphic* property copies the bitmap to the picture object. However, the picture object does not take ownership of the bitmap, so after making the assignment, you must free it.

If you run the application now, you see that client area of the form has a white region, representing the bitmap. If you size the window so that the client area cannot display the entire image, you'll see that the scroll box automatically shows scroll bars to allow display of the rest of the image. But if you try to draw on the image, you don't get any graphics, because the application is still drawing on the form, which is now behind the image and the scroll box.

See Also

Placing the Control (see page 2196)

Drawing On the Bitmap (see page 2188)

3.2.4.13.49 Sharing Code Among Event Handlers

Any time you find that many your event handlers use the same code, you can make your application more efficient by moving the repeated code into a routine that all event handlers can share.

To add a method to a form:

1. Add the method declaration to the form object. You can add the declaration in either the **public** or **private** parts at the end of the form object's declaration. If the code is just sharing the details of handling some events, it's probably safest to make the shared method **private**.
2. Write the method implementation in the implementation part of the form unit.

The header for the method implementation must match the declaration exactly, with the same parameters in the same order.

See Also

Drawing Shapes (see page 2190)

3.2.4.13.50 Tracking Movement

The problem with this example as the *OnMouseMove* event handler is currently written is that it draws the line to the current mouse position from the last *mouse position*, not from the original position. You can correct this by moving the drawing position to the origin point, then drawing to the current point:

```
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
if Drawing then
begin
Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
Canvas.LineTo(X, Y);
end;
```

```
end;
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
    Canvas->LineTo(X, Y);
  }
}
```

The above tracks the current mouse position, but the intermediate lines do not go away, so you can hardly see the final line. The example needs to erase each line before drawing the next one, by keeping track of where the previous one was. The *MovePt* field allows you to do this.

MovePt must be set to the endpoint of each intermediate line, so you can use *MovePt* and *Origin* to erase that line the next time a line is drawn:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
Drawing := True;
Canvas.MoveTo(X, Y);
Origin := Point(X, Y);
MovePt := Point(X, Y);{ keep track of where this move was }
end;
procedure TForm1.FormMouseMove(Sender: TObject;Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
if Drawing then
begin
Canvas.Pen.Mode := pmNotXor;{ use XOR mode to draw/erase }
Canvas.MoveTo(Origin.X, Origin.Y);{ move pen back to origin }
Canvas.LineTo(MovePt.X, MovePt.Y);{ erase the old line }
Canvas.MoveTo(Origin.X, Origin.Y);{ start at origin again }
Canvas.LineTo(X, Y);{ draw the new line }
end;
MovePt := Point(X, Y);{ record point for next move }
Canvas.Pen.Mode := pmCopy;
end;
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
  Origin = Point(X, Y);      // record where the line starts
  MovePt = Point(X, Y);      // record last endpoint
}
void __fastcall TForm1::FormMouseMove(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  if (Drawing)
  {
    Canvas->Pen->Mode = pmNotXor;        // use XOR mode to draw/erase
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
    Canvas->LineTo(MovePt.x, MovePt.y);  // erase old line
    Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point again
    Canvas->LineTo(X, Y);                // draw new line
  }
  MovePt = Point(X, Y);      // record new endpoint
  Canvas->Pen->Mode = pmCopy;
}
```

Now you get a "rubber band" effect when you draw the line. By changing the pen's mode to *pmNotXor*, you have it combine your line with the background pixels. When you go to erase the line, you're actually setting the pixels back to the way they were. By changing the pen mode back to *pmCopy* (its default value) after drawing the lines, you ensure that the pen is ready to do its final

drawing when you release the mouse button.

See Also

Tracking the Origin Point (see page 2204)

3.2.4.13.51 Tracking the Origin Point

When drawing lines, track the point where the line starts with the *Origin* field. *Origin* must be set to the point where the mouse-down event occurs, so the mouse-up event handler can use *Origin* to place the beginning of the line, as in this code:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
Drawing := True;
Canvas.MoveTo(X, Y);
Origin := Point(X, Y);{ record where the line starts }
end;
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
Canvas.MoveTo(Origin.X, Origin.Y);{ move pen to starting point }
Canvas.LineTo(X, Y);
Drawing := False;
end;
void __fastcall TForm1::FormMouseDown(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Drawing = true;            // set the Drawing flag
  Canvas->MoveTo(X, Y);      // set pen position
  Origin = Point(X, Y);      // record where the line starts
}
void __fastcall TForm1::FormMouseUp(TObject *Sender, TMouseButton Button,
  TShiftState Shift, int X, int Y)
{
  Canvas->MoveTo(Origin.x, Origin.y);  // move pen to starting point
  Canvas->LineTo(X, Y);                // draw line from PenPos to (X, Y)
  Drawing = false;                     // clear the Drawing flag
}
```

Those changes get the application to draw the final line again, but they do not draw any intermediate actions—the application does not yet support "rubber banding."

See Also

Tracking Movement (see page 2202)

3.2.4.13.52 Types of Graphic Objects

The component library provides the following graphic objects. These objects have methods to draw on the canvas, which are described in Using Canvas methods to draw graphic objects (see page 2205) and to load and save to graphics files, as described in Loading and saving graphics files (see page 2194)

Graphic object types

Object	Description
Picture	Used to hold any graphic image. To add additional graphic file formats, use the Picture *Register* method. Use this to handle arbitrary files such as displaying images in an image control.
Bitmap	A powerful graphics object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk. Creating copies of a bitmap is fast since the *handle* is copied, not the image.

Clipboard	Represents the container for any text or graphics that are cut, copied, or pasted from or to an application. With the clipboard, you can get and retrieve data according to the appropriate format; handle reference counting, and opening and closing the clipboard; manage and manipulate formats for objects in the clipboard.
Icon	Represents the value loaded from an icon file (::ICO file).
Metafile (VCL applications only)	Contains a file that records the operations required to construct an image, rather than contain the actual bitmap pixels of the image. Metafiles or drawings are extremely scalable without the loss of image detail and often require much less memory than bitmaps, particularly for high-resolution devices, such as printers. However, metafiles and drawings do not display as fast as bitmaps. Use a metafile or drawing when versatility or precision is more important than performance.

See Also

Common Properties and Methods of Canvas (see page 2184)

Refreshing the Screen (see page 2196)

3.2.4.13.53 Using Brushes

The Brush property of a canvas controls the way you fill areas, including the interior of shapes. Filling an area with a brush is a way of changing a large number of adjacent pixels in a specified way.

The brush has three properties you can manipulate:

- Color property changes the fill color.
- Style property changes the brush style.
- Bitmap property uses a bitmap as a brush pattern.

The values of these properties determine the way the canvas fills shapes or other areas. By default, every brush starts out white, with a solid style and no pattern bitmap.

You can use TBrushRecall for quick saving off and restoring the properties of brushes.

See Also

Using Pens (see page 2206)

Reading and Setting Pixels (see page 2196)

3.2.4.13.54 Using Canvas Methods to Draw Graphic Objects

This topic shows how to use some common methods to draw graphic objects. It covers:

- Drawing lines and polylines. (see page 2187)
- Drawing shapes. (see page 2190)
- Drawing rounded rectangles. (see page 2189)
- Drawing polygons. (see page 2189)

See Also

Overview of Graphics Programming (see page 2176)

Using the Properties of the Canvas Object (see page 2207)

Handling Multiple Drawing Objects in Your Application (see page 2192)

Drawing on a Bitmap (see page 2188)

Loading and Saving Graphics Files (see page 2194)

Using the Clipboard with Graphics (see page 2206)

Rubber Banding Example (see page 2199)

3.2.4.13.55 Using Drawing Tools

Now that you can tell what tool to use, you must indicate how to draw the different shapes. The only methods that perform any drawing are the mouse-move and mouse-up handlers, and the only drawing code draws lines, no matter what tool is selected.

To use different drawing tools, your code needs to specify how to draw, based on the selected tool. You add this instruction to each tool's event handler.

This topic describes:

- Drawing shapes (see page 2190).
- Sharing code among event handlers (see page 2202).

See Also

Keeping Track of Which Drawing Tool to Use (see page 2193)

Changing the Tool with Speed Buttons (see page 2184)

3.2.4.13.56 Using Pens

The Pen property of a canvas controls the way lines appear, including lines drawn as the outlines of shapes. Drawing a straight line is really just changing a group of pixels that lie between two points.

The pen itself has four properties you can change:

- Color property changes the pen color.
- Width property changes the pen width.
- Style property changes the pen style.
- Mode property changes the pen mode.

The values of these properties determine how the pen changes the pixels in the line. By default, every pen starts out black, with a width of 1 pixel, a solid style, and a mode called copy that overwrites anything already on the canvas.

You can use TPenRecall for quick saving off and restoring the properties of pens.

See Also

Using Brushes (see page 2205)

Reading and Setting Pixels (see page 2196)

3.2.4.13.57 Using the Clipboard with Graphics

You can use the Windows clipboard to copy and paste graphics within your applications or to exchange graphics with other applications. The VCL's clipboard object makes it easy to handle different kinds of information, including graphics.

Before you can use the clipboard object in your application, you must add the Clipbrd unit to the uses clause of any unit that needs to access clipboard data.

See Also

Overview of Graphics Programming (see page 2176)

Using the Properties of the Canvas Object (see page 2207)

Using Canvas Methods to Draw Graphic Objects (see page 2205)

Handling Multiple Drawing Objects in Your Application (see page 2192)

Drawing on a Bitmap (see page 2188)

Loading and Saving Graphics Files (see page 2194)

Rubber Banding Example (see page 2199)

3.2.4.13.58 Using the Properties of the Canvas Object

With the Canvas object, you can set the properties of a pen for drawing lines, a brush for filling shapes, a font for writing text, and an array of pixels to represent the image.

This topic describes:

- Using pens (see page 2206).
- Using brushes (see page 2205).
- Reading and setting pixels (see page 2196).

See Also

Overview of Graphics Programming (see page 2176)

Using Canvas Methods to Draw Graphic Objects (see page 2205)

Handling Multiple Drawing Objects in Your Application (see page 2192)

Drawing on a Bitmap (see page 2188)

Loading and Saving Graphics Files (see page 2194)

Using the Clipboard with Graphics (see page 2206)

Rubber Banding Example (see page 2199)

3.2.4.13.59 What's in a Mouse Event

A mouse event occurs when a user moves the mouse in the user interface of an application. The VCL has three mouse events.

Mouse events

Event	Description
OnMouseDown event	Occurs when the user presses a mouse button with the mouse pointer over a control.
OnMouseMove event	Occurs when the user moves the mouse while the mouse pointer is over a control.
OnMouseUp event	Occurs when the user releases a mouse button that was pressed with the mouse pointer over a component.

When an application detects a mouse action, it calls whatever event handler you've defined for the corresponding event, passing five parameters. Use the information in those parameters to customize your responses to the events. The five parameters are as follows:

Mouse-event parameters

Parameter	Meaning
Sender	The object that detected the mouse action
Button	Indicates which mouse button was involved: *mbLeft, mbMiddle*, or *mbRight*
Shift	Indicates the state of the `Alt`, `Ctrl`, and `Shift` keys at the time of the mouse action
X, Y	The coordinates where the event occurred

Most of the time, you need the coordinates returned in a mouse-event handler, but sometimes you also need to check *Button* to determine which mouse button caused the event.

Note: Delphi uses the same criteria as Microsoft Windows in determining which mouse button has been pressed. Thus, if you have switched the default "primary" and "secondary" mouse buttons (so that the right mouse button is now the primary button), clicking the primary (right) button will record *mbLeft* as the value of the *Button* parameter.

See Also

Responding to a Mouse-down Action (see page 2198)

Responding to a Mouse-up Action (see page 2199)

Responding to a Mouse Move (see page 2198)

3.2.4.13.60 Working with Graphics and Multimedia: Overview

Graphics and multimedia elements can add polish to your applications. You can introduce these features into your application in a variety of ways. To add graphical elements, you can insert pre-drawn pictures at design time, create them using graphical controls at design time, or draw them dynamically at runtime. To add multimedia capabilities, you can use special components that can play audio and video clips.

This following topics describe how to enhance your applications by introducing graphics or multimedia elements:

- Overview of Graphics Programming (see page 2176)
- Working with multimedia (see page 2177)

3.2.4.14 Working with packages and components

Topics

Name	Description
Loading Packages with the LoadPackage Function (see page 2211)	You can also load a package at runtime by calling the *LoadPackage* function. *LoadPackage* loads the package specified by its name parameter, checks for duplicate units, and calls the initialization blocks of all units contained in the package. For example, the following code could be executed when a file is chosen in a file-selection dialog.
Working with Packages and Components: Overview (see page 2211)	A *package* is a special dynamic-link library used by applications, the IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time* packages are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by calling runtime packages. To distinguish them from other DLLs, package libraries are stored in files that end with the .bpl (Borland package library) extension. Like other runtime libraries, packages contain code that can be shared among applications. For... more (see page 2211)
Compiling Packages (see page 2212)	You can compile a package from the IDE or from the command line.

Creating and Editing Packages (see page 2213)	Creating a package involves specifying: • A *name* for the package. • A list of other packages to be *required* by, or linked to, the new package. • A list of unit files to be *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units . The Contains clause is where you put the source-code units for custom components that you want to compile into a package. RAD Studio generates a package source file (.dpk). • Creating a Package (see page 2213) • Editing an Existing Package (see page 2216) • Editing Package Source Files Manually (see page 2216) • Understanding the Structure of a (see page 2220)... more (see page 2213)
Creating a Package (see page 2213)	Refer to Understanding the structure of a package (see page 2220) for more information about the steps outlined here.
Custom Packages (see page 2214)	A custom package is either a .bpl you code and compile yourself or an existing package from a third-party vendor. To use a custom runtime package with an application, choose **Project** ▶ **Options** and add the name of the package to the Runtime Packages edit box on the Packages page. For example, suppose you have a statistical package called stats.bpl. To use it in an application, the line you enter in the Runtime Packages edit box might look like this:
Deciding Which Runtime Packages to Use (see page 2214)	Several runtime packages, including rtl and vcl (VCL application) , supply basic language and component support. The vcl (VCL) package contains the most commonly used components; the rtl package includes all the non-component system functions and Windows interface elements. It does not include database or other special components, which are available in separate packages. To create a client/server database application that uses packages, you need several runtime packages, including vcl, vcldb, rtl, and dbrtl (VCL) . If you want to use visual components in your application, you also need vclx (VCL) . To use these packages, choose **Project** ▶ **Options**,... more (see page 2214)
Deploying Packages (see page 2215)	You deploy packages much like you deploy other applications. The files you distribute with a deployed package may vary. The bpl and any packages or dlls required by the bpl must be distributed. ***Files deployed with a package***
Design-time Packages (see page 2215)	Design-time packages are used to install components on the IDE's **Tool palette** and to create special property editors for custom components. Which ones are installed depends on which edition of Delphi you are using and whether or not you have customized it. You can view a list of what packages are installed on your system by choosing **Component** ▶ **Install Packages...**. The design-time packages work by calling runtime packages, which they reference in their Requires clause (see page 2220). For example, dclstd references vcl. The dclstd itself contains additional functionality that makes many of the standard components available on the **Tool palette... more (see page 2215)**
Editing an Existing Package (see page 2216)	
Editing Package Source Files Manually (see page 2216)	Package source files, like project files, are generated by Delphi from information you supply. Like project files, they can also be edited manually. A package source file should be saved with the .dpk (Delphi package) extension to avoid confusion with other files containing Del source code.
Installing Component Packages (see page 2217)	All components are installed in the IDE as packages. If you've written your own components, create and compile (see page 2213) a package that contains them. Your component source code must follow the model described in Overview of component creation. (see page 1313)

Package Collection Files (see page 2217)	Package collections (.dpc files) offer a convenient way to distribute packages to other developers. Each package collection contains one or more packages, including bpls and any additional files you want to distribute with them. When a package collection is selected for IDE installation, its constituent files are automatically extracted from their .pce container; the Installation dialog box offers a choice of installing all packages in the collection or installing packages selectively.
Package Files Created by Compiling (see page 2218)	To create a package, you compile a source file that has a .dpk extension. The base name of the .dpk file becomes the base name of the files generated by the compiler. For example, if you compile a package source file called traypak.dpk, the compiler creates a package called traypak.bpl.
Packages and Standard DLLs (see page 2219)	Create a package when you want to make a custom component that's available through the IDE. Create a standard DLL when you want to build a library that can be called from any application, regardless of the development tool used to build the application. The following table lists the file types associated with packages: ***Package files***
Package-specific Compiler Directives (see page 2219)	The following table lists package-specific compiler directives that you can insert into your source code. ***Package-specific compiler directives***
Runtime Packages (see page 2220)	Runtime packages are deployed with your applications. They provide functionality when a user runs the application. To run an application that uses packages, a computer must have both the application's executable file and all the packages (.bpl files) that the application uses. The .bpl files must be on the system path for an application to use them. When you deploy an application, you must make sure that users have correct versions of any required .bpls. • Loading packages in an application (see page 2221) • Deciding which runtime packages to use (see page 2214) • Custom packages (see page 2214)
Understanding the Structure of a Package (see page 2220)	Packages include the following parts: • Package name (see page 2220) • Requires clause (see page 2220) • Contains clause (see page 2220)
Loading Packages in an Application (see page 2221)	You can dynamically load packages by either: • Choosing Project Options dialog box in the IDE; or • Using the LoadPackage function (see page 2211).
Compiling and Linking from the Command Line (see page 2222)	When you compile from the command line, you can use the package-specific switches listed in the following table. ***Package-specific command-line compiler switches***
Weak Packaging (see page 2223)	The **$WEAKPACKAGEUNIT** directive affects the way a .dcu file is stored in a package's .dcp and .bpl files. (For information about files generated by the compiler, see Package files created when compiling (see page 2218).) If **{$WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from bpls when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be *weakly packaged*. For example, suppose you've created a package called pack1 that contains only one unit, unit1. Suppose unit1 does not... more (see page 2223)
Why Use Packages? (see page 2224)	Design-time packages simplify the tasks of distributing and installing custom components. Runtime packages, which are optional, offer several advantages over conventional programming. By compiling reused code into a runtime library, you can share it among applications. For example, all of your applications—including Delphi itself—can access standard components through packages. Since the applications don't have separate copies of the component library bound into their executables, the executables are much smaller, saving both system resources and hard disk storage. Moreover, packages allow faster compilation because only code unique to the application is compiled with each build.

3.2.4.14.1 Loading Packages with the LoadPackage Function

You can also load a package at runtime by calling the *LoadPackage* function. *LoadPackage* loads the package specified by its name parameter, checks for duplicate units, and calls the initialization blocks of all units contained in the package. For example, the following code could be executed when a file is chosen in a file-selection dialog.

```
with OpenDialog1 do
  if Execute then
    with PackageList.Items do
      AddObject(FileName, Pointer(LoadPackage(FileName)));
if (OpenDialog1->Execute())
  PackageList->Items->AddObject(OpenDialog1->FileName, (TObject
*)LoadPackage(OpenDialog1->FileName));
```

To unload a package dynamically, call *UnloadPackage*. Be careful to destroy any instances of classes defined in the package and to unregister classes that were registered by it.

See Also

Working with Packages and Components: Overview (see page 2211)

Runtime Packages (see page 2220)

Deploying Packages (see page 2215)

3.2.4.14.2 Working with Packages and Components: Overview

A *package* is a special dynamic-link library used by applications, the IDE, or both. *Runtime packages* provide functionality when a user runs an application. *Design-time* packages are used to install components in the IDE and to create special property editors for custom components. A single package can function at both design time and runtime, and design-time packages frequently work by calling runtime packages. To distinguish them from other DLLs, package libraries are stored in files that end with the .bpl (Borland package library) extension.

Like other runtime libraries, packages contain code that can be shared among applications. For example, the most frequently used VCL components reside in a package called vcl . Each time you create a new default VCL application, it automatically uses vcl. When you compile an application created this way, the application's executable image contains only the code and data unique to it; the common code is in the runtime package called vcl90.bpl. A computer with several package-enabled applications installed on it needs only a single copy of vcl90.bpl, which is shared by all the applications and the IDE itself.

Several runtime packages encapsulate VCL components while several design-time packages manipulate components in the IDE.

You can build applications with or without packages. However, if you want to add custom components to the IDE, you must install them as design-time packages.

You can create your own runtime packages to share among applications. If you write Delphi components, you can compile your components into design-time packages before installing them.

See Also

Overview of Component Creation (see page 1313)

Why Use Packages? (see page 2224)

Packages and Standard DLLs (see page 2219)

Runtime Packages (see page 2220)

Loading Packages in an Application (see page 2221)

Loading Packages with the LoadPackage Function (see page 2211)

Deciding Which Runtime Packages to Use (see page 2214)
Custom Packages (see page 2214)
Design-time Packages (see page 2215)
Installing Component Packages (see page 2217)
Creating and Editing Packages (see page 2213)
Creating a Package (see page 2213)
Editing an Existing Package (see page 2216)
Understanding the Structure of a Package (see page 2220)
Naming Packages (see page 2220)
Requires Clause (see page 2220)
Avoiding Circular Package References (see page 2220)
Handling Duplicate Package References (see page 2220)
Contains Clause (see page 2220)
Avoiding Redundant Source Code Uses (see page 2220)
Editing Package Source Files Manually (see page 2216)
Compiling Packages (see page 2212)
Package-specific Compiler Directives (see page 2219)
Weak Packaging (see page 2223)
Compiling and Linking From the Command Line (see page 2222)
Package Files Created by Compiling (see page 2218)
Deploying Packages (see page 2215)
Deploying Applications that Use Packages (see page 2215)
Distributing Packages to Other Developers (see page 2215)
Package Collection Files (see page 2217)

3.2.4.14.3 Compiling Packages

You can compile a package from the IDE or from the command line.

To recompile a package by itself from the IDE

1. Choose **File** ▸ **Open** and select a package (.dpk or .dpkw).
2. Click Open.
3. When the package opens:

- In the **Project Manager**, right-click the package and choose Compile.
- In the IDE, choose **Project** ▸ **Build**.

Note: Right-click the package project nodes for options to compile or build.

You can insert compiler directives into your package source code.

If you compile from the command line, you can use several package-specific switches.

- Package-specific Compiler Directives (see page 2219)
- Weak Packaging (see page 2223)
- Using the Command-line Compiler and Linker (see page 2222)
- Package Files Created by a Successful Compilation (see page 2218)

See Also

Working with Packages and Components: Overview (see page 2211)

Creating and Editing Packages (see page 2213)

3.2.4.14.4 Creating and Editing Packages

Creating a package involves specifying:

- A *name* for the package.
- A list of other packages to be *required* by, or linked to, the new package.
- A list of unit files to be *contained* by, or bound into, the package when it is compiled. The package is essentially a wrapper for these source-code units . The Contains clause is where you put the source-code units for custom components that you want to compile into a package.

RAD Studio generates a package source file (.dpk).

- Creating a Package (see page 2213)
- Editing an Existing Package (see page 2216)
- Editing Package Source Files Manually (see page 2216)
- Understanding the Structure of a Package (see page 2220)
- Compiling packages (see page 2212)

See Also

Working with Packages and Components: Overview (see page 2211)

3.2.4.14.5 Creating a Package

Refer to Understanding the structure of a package (see page 2220) for more information about the steps outlined here.

To create a package

1. Choose **File** ▸ **New** ▸ **Other**, select the Package icon under Delphi Projects, and click OK. The generated package appears in the **Project Manager**. The **Project Manager** displays a *Requires* node and a *Contains* node for the new package.
2. To add a unit to the **contains** clause, right-click the contains node in the **Project Manager** and select Add. In the Add Unit page, type a .pas file name in the Unit file name edit box, or click Browse to browse for the file, and then click OK. The unit you've selected appears under the Contains node in the **Project Manager**. You can add additional units by repeating this step.
3. To add a package to the **requires** clause, right-click the requires node in the **Project Manager** and select Add Reference. In the Requires page, type a .dcp file name in the Package name edit box, or click Browse to browse for the file, and then click OK. The package you've selected appears under the Requires node in the **Project Manager**. You can add additional packages by repeating this step.
4. In the **Project Manager**, right-click your package and select Compile.

See Also

Working with Packages and Components: Overview (see page 2211)

Creating and Editing Packages (see page 2213)

Editing an Existing Package (see page 2216)

3.2.4.14.6 Custom Packages

A custom package is either a .bpl you code and compile yourself or an existing package from a third-party vendor. To use a custom runtime package with an application, choose **Project** ▸ **Options** and add the name of the package to the Runtime Packages edit box on the Packages page.

For example, suppose you have a statistical package called stats.bpl. To use it in an application, the line you enter in the Runtime Packages edit box might look like this:

```
vcl;rtl;vcldb;stats //In VCL applications
```

If you create your own packages, add them to the list as needed.

See Also

Working with Packages and Components: Overview (see page 2211)

Runtime Packages (see page 2220)

Loading Packages in an Application (see page 2221)

3.2.4.14.7 Deciding Which Runtime Packages to Use

Several runtime packages, including rtl and vcl (VCL application) , supply basic language and component support. The vcl (VCL) package contains the most commonly used components; the rtl package includes all the non-component system functions and Windows interface elements. It does not include database or other special components, which are available in separate packages.

To create a client/server database application that uses packages, you need several runtime packages, including vcl, vcldb, rtl, and dbrtl (VCL) . If you want to use visual components in your application, you also need vclx (VCL) . To use these packages, choose **Project** ▸ **Options**, select the Packages tab, and make sure the following list is included in the Runtime Packages edit box. You need netclx for Web server applications, as well as baseclx and probably visualclx.

```
vcl;rtl;vcldb;vclx; //For VCL applications
```

Note: In VCL applications, you don't have to manually include vcl and rtl, because they are referenced in the Requires clause (see page 2220) of vcldb. Your application compiles just the same whether or not vcl and rtl are included in the Runtime Packages edit box.

Another way you can determine which packages are called by an application is to run it then review the event log (choose **View** ▸ **Debug Windows** ▸ **Event Log**). The event log displays every module that is loaded including all packages. The full package names are listed. So, for example, for vcl90.bpl , you would see a line similar to the following in a VCL application:

```
Module Load: vcl90.bpl Has Debug Info. Base Address $400B0000. Process Project1.exe ($22C)
```

See Also

Working with Packages and Components: Overview (see page 2211)

Runtime Packages (see page 2220)

Loading Packages in an Application (see page 2221)

Custom Packages (see page 2214)

3.2.4.14.8 Deploying Packages

You deploy packages much like you deploy other applications. The files you distribute with a deployed package may vary. The bpl and any packages or dlls required by the bpl must be distributed.

Files deployed with a package

File	Description
ComponentName.h	Allows end users access to the class interfaces.
ComponentName.cpp	Allows end users access to the component source.
.bpi, .obj, and .lib	Allows end users to link applications.

For general deployment information, refer to Deploying applications. (see page 1947)

Deploying applications that use packages

When distributing an application that uses runtime packages, make sure that your users have the application's .exe file as well as all the library (.bpl or .dll) files that the application calls. If the library files are in a different directory from the .exe file, they must be accessible through the user's Path. You may want to follow the convention of putting library files in the Windows\System directory. If you use InstallAware Express, your installation script can check the user's system for any packages it requires before blindly reinstalling them.

Distributing packages to other developers

If you distribute runtime or design-time packages to other Delphi developers, be sure to supply both .dcp and .bpl files. You will probably want to include .dcu files as well.

See Also

Working with Packages and Components: Overview (see page 2211)

3.2.4.14.9 Design-time Packages

Design-time packages are used to install components on the IDE's **Tool palette** and to create special property editors for custom components. Which ones are installed depends on which edition of Delphi you are using and whether or not you have customized it. You can view a list of what packages are installed on your system by choosing **Component ▸ Install Packages...**.

The design-time packages work by calling runtime packages, which they reference in their Requires clause (see page 2220). For example, dclstd references vcl. The dclstd itself contains additional functionality that makes many of the standard components available on the **Tool palette**.

In addition to preinstalled packages, you can install your own component packages, or component packages from third-party developers, in the IDE. The dclusr design-time package is provided as a default container for new components. See Installing Component Packages (see page 2217)

C++Builder supports compiling design-time packages that contain Delphi source files. However, if any of those Delphi sources make reference to IDE-supplied design-time units such as DesignIntf, DesignEditors, and ToolsAPI that exist in DesignIDE100.bpl, you must take steps to ensure that the references can be resolved by the C++Builder package. See Compiling C++ Design-Time Packages That Contain Delphi Source for details.

See Also

Working with Packages and Components: Overview (see page 2211)

Runtime Packages (see page 2220)

Creating and Editing Packages (see page 2213)

Compiling C++ Design-Time Packages That Contain Delphi Source

3.2.4.14.10 Editing an Existing Package

To edit an existing package:

1. Choose **File** ▶ **Open** (or **File** ▶ **Reopen**) and select a dpk file.
2. In the Project Manager, select one of the packages in the Requires node, right-click, and choose Open.

The Project Options dialog has a Default check box in the lower left corner. If you click OK when this box is checked, the options you've chosen are saved as default settings for new projects. To restore the original defaults, delete or rename the defproj.dof file.

See Also

Working with Packages and Components: Overview (see page 2211)

Creating and Editing Packages (see page 2213)

Creating a Package (see page 2213)

Understanding the Structure of a Package (see page 2220)

Compiling Packages (see page 2212)

3.2.4.14.11 Editing Package Source Files Manually

Package source files, like project files, are generated by Delphi from information you supply. Like project files, they can also be edited manually. A package source file should be saved with the .dpk (Delphi package) extension to avoid confusion with other files containing Del source code.

To open a package source file in the Code editor

1. Open the package in RAD Studio.
2. Right-click the package in the **Project Manager** and choose View Source.

- The **package** heading specifies the name for the package.
- The **requires** clause lists other, external packages used by the current package. If a package does not contain any units that use units in another package, then it doesn't need a requires clause.
- The **contains** clause identifies the unit files to be compiled and bound into the package. All units used by contained units which do not exist in required packages will also be bound into the package, although they won't be listed in the contains clause (the compiler will give a warning). For example, the following VCL code declares the vcldb package (in the source file vcldb90.bpl):

```
package MyPack;
{$R *.res}
    ...{compiler directives omitted}
requires
    rtl,
    vcl;
contains
    Db,
    NewComponent1 in 'NewComponent1.pas';
end.
```

See Also

Working with Packages and Components: Overview (see page 2211)

Creating and Editing Packages (see page 2213)

Understanding the Structure of a Package (see page 2220)

Compiling Packages (see page 2212)

3.2.4.14.12 Installing Component Packages

All components are installed in the IDE as packages. If you've written your own components, create and compile (see page 2213) a package that contains them. Your component source code must follow the model described in Overview of component creation. (see page 1313)

To install or uninstall your own components, or components from a third-party vendor

1. If you are installing a new package, copy or move the package files to a local directory. If the package is shipped with .bpl (see page 2219), be sure to copy all of them. The directory where you store the .dcp file—and the .dcu files, if they are included with the distribution—must be in the Delphi Library Path.
2. Choose **Component ▸ Install Packages** from the IDE menu, or choose **Project ▸ Options** and click the Packages tab. A list of available packages appears in the Design packages list box.

- To install a package in the IDE, select the check box next to it.
- To uninstall a package, uncheck its check box.
- To see a list of components included in an installed package, select the package and click Components.
- To add a package to the list, click Add and browse in the Add Design Package dialog for the directory where the .bpl file resides (see step 1). Select the .bpl or .dpc file and click Open. If you select a .dpc file, a new dialog box appears to handle the extraction of the .bpl and other files from the package collection.
- To remove a package from the list, select the package and click Remove.

3. Click OK.

The components in the package are installed on the **Tool palette** pages specified in the components' *RegisterComponents* procedure, with the names they were assigned in the same procedure.

New projects are created with all available packages installed, unless you change the ddefault settings. To make the current installation choices into the automatic default for new projects, check the Default check box at the bottom of the Packages tab of the Project Options dialog box.

To remove components from the **Tool palette** without uninstalling a package, right-click the component to invoke the context menu and choose Hide "<ComponentName>" Button.

See Also

Working with Packages and Components: Overview (see page 2211)

Design-time Packages (see page 2215)

3.2.4.14.13 Package Collection Files

Package collections (.dpc files) offer a convenient way to distribute packages to other developers. Each package collection contains one or more packages, including bpls and any additional files you want to distribute with them. When a package collection is selected for IDE installation, its constituent files are automatically extracted from their .pce container; the Installation dialog box offers a choice of installing all packages in the collection or installing packages selectively.

To create package collection files

1. Choose **Tools ▸ Package Collection Editor** to open the Package Collection editor.
2. Either choose **Edit ▸ Add Package** or click the Add a package button, then select a bpl in the Select Package dialog and click Open. To add more bpls to the collection, click the Add a package button again. A tree diagram on the left side of the Package editor displays the bpls as you add them. To remove a package, select it and either choose **Edit ▸ Remove Package** or click

the Remove the selected package button.

3. Select the Collection node at the top of the tree diagram. On the right side of the Package Collection editor, two fields appear:

- In the Author/Vendor Name edit box, you can enter optional information about your package collection that appear in the Installation dialog when users install packages.
- Under Directory list, list the default directories where you want the files in your package collection to be installed. Use the Add, Edit, and Delete buttons to edit this list. For example, suppose you want all source code files to be copied to the same directory. In this case, you might enter Source as a Directory name with C:\MyPackage\Source as the Suggested path. The Installation dialog box will display C:\MyPackage\Source as the suggested path for the directory.

4. In addition to bpls, your package collection can contain .dcp, .dcu, and .pas (unit) files, documentation, and any other files you want to include with the distribution. Ancillary files are placed in file groups associated with specific packages (bpls); the files in a group are installed only when their associated bpl is installed. To place ancillary files in your package collection, select a bpl in the tree diagram and click the Add a file group button; type a name for the file group. Add more file groups, if desired, in the same way. When you select a file group, new fields will appear on the right in the Package Collection editor.

- In the Install Directory list box, select the directory where you want files in this group to be installed. The drop-down list includes the directories you entered under Directory list in step 3, above.
- Check the Optional Group check box if you want installation of the files in this group to be optional.
- Under Include Files, list the files you want to include in this group. Use the Add, Delete, and Auto buttons to edit the list. The Auto button allows you to select all files with specified extensions that are listed in the **contains** clause of the package; the Package Collection editor uses the global Library Path to search for these files.

5. You can select installation directories for the packages listed in the **requires** clause of any package in your collection. When you select a bpl in the tree diagram, four new fields appear on the right side of the Package Collection editor:

- In the Required Executables list box, select the directory where you want the .bpl files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory list in step 3, above.) The Package Collection editor searches for these files using Delphi's global Library Path and lists them under Required Executable Files.
- In the Required Libraries list box, select the directory where you want the .dcp files for packages listed in the **requires** clause to be installed. (The drop-down list includes the directories you entered under Directory List in step 3, above.) The Package Collection editor searches for these files using the global Library Path and lists them under Required Library Files.

6. To save your package collection source file, choose **File ▸ Save**. Package collection source files should be saved with the .pce extension.

7. To build your package collection, click the Compile button. The Package Collection editor generates a .dpc file with the same name as your source (.pce) file. If you have not yet saved the source file, the editor queries you for a file name before compiling.

To edit or recompile an existing .pce file, select **File ▸ Open** in the Package Collection editor and locate the file you want to work with.

See Also

Working with Packages and Components: Overview (see page 2211)

3.2.4.14.14 Package Files Created by Compiling

To create a package, you compile a source file that has a .dpk extension. The base name of the .dpk file becomes the base name of the files generated by the compiler. For example, if you compile a package source file called traypak.dpk, the compiler creates a package called traypak.bpl.

```
<PROJECT value="Traypak.bpl"/>
```

A successfully compiled package includes .dcp, .dcu and bpl files. For a detailed description of these files, see Packages and standard DLLs (see page 2219).

These files are generated by default in the directories specified in Library page of the **Tools ▸ Options ▸ Environment Options ▸ Delphi Options ▸ Library** dialog. You can override the default settings by right-clicking the package in the **Project**

Manager and choosing Options to display the Project Options dialog; make any changes on the Directories/Conditionals page.

See Also

Working with Packages and Components: Overview (see page 2211)

Compiling Packages (see page 2212)

Compiling and Linking from the Command Line (see page 2222)

3.2.4.14.15 Packages and Standard DLLs

Create a package when you want to make a custom component that's available through the IDE. Create a standard DLL when you want to build a library that can be called from any application, regardless of the development tool used to build the application.

The following table lists the file types associated with packages:

Package files

File extension	Contents
.bpf	A source file required for a package.
bpi	A Borland package import library. A .bpi is created for each package. The bpis for bpls is analogous to import libraries for dlls. This file is passed to the linker by applications using the package to resolve references to functions in the package. The base name for the bpi is the base name for the package source file.
bpk and bpkw	The project options source file. This file is the XML portion of the package project. The ProjectName.bpk and ProjectName.cpp combined are used to manage settings, options, and files used by the package project. .bpk and .bpkw packages are identical, but use the .bpkw extension for packages that you want to use in cross-platform applications.
bpl	The runtime package. This file is a Windows .dll with special -specific features. The base name for the .bpl is the base name of the .bpk or .bpkw source file.
cpp	ProjectName.cpp contains the entry point for the package. Additionally, each component contained within the package generally resides within a .cpp file.
h	The header file or interface for the component. C omponentN ame.h is the companion to C omponentN ame.cpp.
lib	A static library, or collection of .objs , used in place of a .bpi when the application does not use runtime packages. Generated only if the **-Gl** (Generate .lib file) linker option is selected.
obj	A binary image for a unit file contained in a package. One .obj is created, when necessary, for each .cpp file.

Note: Packages share their global data with other modules in an application.

See Also

Working with Packages and Components: Overview (see page 2211)

Overview of Component Creation (see page 1313)

3.2.4.14.16 Package-specific Compiler Directives

The following table lists package-specific compiler directives that you can insert into your source code.

Package-specific compiler directives

Directive	Purpose
#pragma package(smart_init)	Assures that packaged units are initialized in the order determined by their dependencies. (Included by default in package source file.)
#pragma package(smart_init, weak)	Packages unit "weakly." See (Put directive in unit source file.)

Note: Including **{$DENYPACKAGEUNIT ON}** in your source code prevents the unit file from being packaged. Including **{$G-}** or **{$IMPORTEDDATA OFF}** may prevent a package from being used in the same application with other packages. Packages compiled with the **{$DESIGNONLY ON}** directive should not ordinarily be used in applications, since they contain extra code required by the IDE. Other compiler directives may be included, if appropriate, in package source code. See Compiler Directives for information on compiler directives not discussed here.

See Package-specific Compiler Directives.

Refer to Creating Packages and DLLs (see page 1882) for additional directives that can be used in all libraries.

See Also

Working with Packages and Components: Overview (see page 2211)

Compiling Packages (see page 2212)

Creating and Editing Packages (see page 2213)

3.2.4.14.17 Runtime Packages

Runtime packages are deployed with your applications. They provide functionality when a user runs the application.

To run an application that uses packages, a computer must have both the application's executable file and all the packages (.bpl files) that the application uses. The .bpl files must be on the system path for an application to use them. When you deploy an application, you must make sure that users have correct versions of any required .bpls.

- Loading packages in an application (see page 2221)
- Deciding which runtime packages to use (see page 2214)
- Custom packages (see page 2214)

See Also

Working with Packages and Components: Overview (see page 2211)

3.2.4.14.18 Understanding the Structure of a Package

Packages include the following parts:

- Package name
- Requires clause
- Contains clause

Naming packages

Package names must be unique within a project. If you name a package Stats, RAD Studio generates a source file for it called Stats.dpk; the compiler generates an executable and a binary image called Stats.bpl and Stats.dcp, respectively. Use Stats to refer to the package in the **requires** clause of another package, or when using the package in an application.

Requires clause

The **requires** clause specifies other, external packages that are used by the current package. An external package included in

the **requires** clause is automatically linked at compile time into any application that uses both the current package and one of the units contained in the external package.

If the unit files contained in your package make references to other packaged units, the other packages should appear in your package's **requires** clause or you should add them. If the other packages are omitted from the **requires** clause, the compiler will import them into your package 'implicitly contained units.'

Note: Most packages that you create require rtl. If using VCL components, you'll also need to include the vcl package.

Avoiding circular package references

Packages cannot contain circular references in their requires clause. This means that:

- A package cannot reference itself in its own **requires** clause.
- A chain of references must terminate without rereferencing any package in the chain. If package A requires package B, then package B cannot require package A; if package A requires package B and package B requires package C, then package C cannot require package A.

Handling duplicate package references

Duplicate references in a package's **requires** clause—or the Runtime Packages edit box—are ignored by the compiler. For programming clarity and readability, however, you should catch and remove duplicate package references.

Contains clause

The **contains** clause identifies the unit files to be bound into the package. If you are writing your own package, put your source code in pas files and include them in the **contains** clause.

Avoiding redundant source code uses

A package cannot appear in the containsclause of another package.

All units included directly in a package's **contains** clause, or included indirectly in any of those units, are bound into the package at compile time.

A unit cannot be contained (directly or indirectly) in more than one package used by the same application, including the IDE. This means that if you create a package that contains one of the units in vcl (VCL) you won't be able to install your package in the IDE. To use an already-packaged unit file in another package, put the first package in the second package's **requires** clause.

See Also

Working with Packages and Components: Overview (see page 2211)

Creating and Editing Packages (see page 2213)

3.2.4.14.19 Loading Packages in an Application

You can dynamically load packages by either:

- Choosing Project Options dialog box in the IDE; or
- Using the LoadPackage function (see page 2211).

To load packages using the >Project>Options dialog box

1. Load or create a project in the IDE.
2. Choose **Project** ▸ **Options**.
3. Choose the Packages tab.
4. Select the Build with Runtime Packages check box, and enter one or more package names in the edit box underneath. Each package is loaded implicitly only when it is needed (that is, when you refer to an object defined in one of the units in that package). (Runtime packages associated with installed design-time packages are already listed in the edit box.)

5. To add a package to an existing list, click the Add button and enter the name of the new package in the Add Runtime Package dialog. To browse from a list of available packages, click the Add button, then click the Browse button next to the Package Name edit box in the Add Runtime Package dialog. If you edit the Search Path edit box in the Add Runtime Package dialog, you can change the global Library Path. You do not need to include file extensions with package names (or the version number representing the Delphi release); that is, vcl90.bpl in a VCL application is written as vcl. If you type directly into the Runtime Package edit box, be sure to separate multiple names with semicolons. For example:

```
rtl;vcl;vcldb;vclado;

vclbde;
```

Packages listed in the Runtime Packages edit box are automatically linked to your application when you compile. Duplicate package names are ignored, and if the Build with runtime packages check box is unchecked, the application is compiled without packages. Runtime packages are selected for the current project only. To make the current choices into automatic defaults for new projects, select the Defaults check box at the bottom of the dialog.

Note: When you create an application with packages, you must include the names of the original Delphi units in the **uses** clause of your source files. For example, the source file for your main form might begin like this:

```
unit MainForm;
interface
uses
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs;

#include "vcldb.h"
```

The units referenced in this VCL example are contained in the vcl and rtl packages. Nonetheless, you must keep these references in the **uses** clause, even if you use vcl and rtl in your application, or you will get compiler errors. In generated source files, the Form Designer adds these units to the uses clause automatically.

See Also

Deciding Which Runtime Packages to Use (see page 2214)

Working with Packages and Components: Overview (see page 2211)

Deploying Packages (see page 2215)

Loading Packages with the LoadPackage Function (see page 2211)

3.2.4.14.20 Compiling and Linking from the Command Line

When you compile from the command line, you can use the package-specific switches listed in the following table.

Package-specific command-line compiler switches

Switch	Purpose
$G	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
-LE*path*	Specifies the directory where the package file (.bpl) will be placed.
-LN*path*	Specifies the directory where the package file (.dcp) will be placed.
-LUpackage	Use packages.
-Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

Note: Using the **-$G-** switch may prevent a package from being used in the same application with other packages. Other

command-line options may be used, if appropriate, when compiling packages. See The Command-line Compiler for information on command-line options not discussed here.

Package-specific command-line compiler and linker switches

Switch	Purpose
tP	Generates a project as a package (compiler switch).
-D "*description*"	Saves the specified description with the package.
-Gb	Generates the .bpl filename.
-Gi	Saves the generated .bpi file. Included by default in package project files.
-Gpd	Generates a design-time-only package.
-Gpr	Generates a runtime-only package.
-Gl	Generates a .lib file.
-Tpp	Builds the project as a package. Included by default in package project files.

See Also

Working with Packages and Components: Overview (see page 2211)

Compiling Packages (see page 2212)

3.2.4.14.21 Weak Packaging

The **$WEAKPACKAGEUNIT** directive affects the way a .dcu file is stored in a package's .dcp and .bpl files. (For information about files generated by the compiler, see Package files created when compiling (see page 2218).) If **{$WEAKPACKAGEUNIT ON}** appears in a unit file, the compiler omits the unit from bpls when possible, and creates a non-packaged local copy of the unit when it is required by another application or package. A unit compiled with this directive is said to be *weakly packaged*.

For example, suppose you've created a package called pack1 that contains only one unit, unit1. Suppose unit1 does not use any additional units, but it makes calls to rare.dll. If you put the **{$WEAKPACKAGEUNIT ON}** directive in unit1.pas (Delphi) or unit1.cpp (C++) when you compile your package, unit1 will not be included in pack1.bpl; you will not have to distribute copies of rare.dll with pack1. However, unit1 will still be included in pack1.dcp. If unit1 is referenced by another package or application that uses pack1, it will be copied from pack1.dcp and compiled directly into the project.

Now suppose you add a second unit, unit2, to pack1. Suppose that unit2 uses unit1. This time, even if you compile pack1 with **{$WEAKPACKAGEUNIT ON}** in unit1.pas, the compiler will include unit1 in pack1.bpl. But other packages or applications that reference unit1 will use the (non-packaged) copy taken from pack1.dcp.

Note: Unit files containing the **{$WEAKPACKAGEUNIT ON}** directive must not have global variables, initialization sections, or finalization sections.

The **{$WEAKPACKAGEUNIT ON}** directive is an advanced feature intended for developers who distribute their packages to other programmers. It can help you to avoid distribution of infrequently used DLLs, and to eliminate conflicts among packages that may depend on the same external library.

For example, the PenWin unit references PenWin.dll. Most projects don't use PenWin, and most computers don't have PenWin.dll installed on them. For this reason, the PenWin unit is weakly packaged in vcl. When you compile a project that uses PenWin and the vcl package, PenWin is copied from vcl70.dcp and bound directly into your project; the resulting executable is statically linked to PenWin.dll.

If PenWin were not weakly packaged, two problems would arise. First, vcl itself would be statically linked to PenWin.dll, and so you could not load it on any computer which didn't have PenWin.dll installed. Second, if you tried to create a package that contained PenWin, a compiler error would result because the PenWin unit would be contained in both vcl and your package. Thus, without weak packaging, PenWin could not be included in standard distributions of vcl.

See Also

Working with Packages and Components: Overview (see page 2211)

Compiling Packages (see page 2212)

Package-specific Compiler Directives (see page 2219)

3.2.4.14.22 Why Use Packages?

Design-time packages simplify the tasks of distributing and installing custom components. Runtime packages, which are optional, offer several advantages over conventional programming. By compiling reused code into a runtime library, you can share it among applications. For example, all of your applications—including Delphi itself—can access standard components through packages. Since the applications don't have separate copies of the component library bound into their executables, the executables are much smaller, saving both system resources and hard disk storage. Moreover, packages allow faster compilation because only code unique to the application is compiled with each build.

See Also

Working with Packages and Components: Overview (see page 2211)

Packages and Standard DLLs (see page 2219)

Overview of Component Creation (see page 1313)

3.2.4.15 Writing multi-threaded applications

Topics

Name	Description
Coordinating threads (see page 2225)	
Writing Multi-threaded Applications (see page 2231)	Several objects make writing multi-threaded applications easier. Multi-threaded applications are applications that include several simultaneous paths of execution. While using multiple threads requires careful thought, it can enhance your programs by: • **Avoiding bottlenecks.** With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process. • **Organizing program behavior.** Often, a program's behavior can... more (see page 2231)
Defining thread objects (see page 2231)	
Executing thread objects (see page 2238)	

3.2.4.15.1 Coordinating threads

Topics

Name	Description
Other Techniques for Sharing Memory (see page 2226)	When using VCL objects, use the main thread to execute your code. Using the main thread ensures that the object does not indirectly access any memory that is also used by VCL objects in other threads. See Using the Main VCL Thread (see page 2235) for more information on the main thread. If the global memory does not need to be shared by multiple threads, consider using thread-local variables instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads. See Using Thread-local Variables (see page 2237) for more information about thread-local variables.
Avoiding Simultaneous Access (see page 2226)	To avoid clashing with other threads when accessing global objects or variables, you may need to block the execution of other threads until your thread code has finished an operation. Be careful not to block other execution threads unnecessarily. Doing so can cause performance to degrade seriously and negate most of the advantages of using multiple threads. Three techniques prevent other threads from accessing the same memory as your thread: • Locking Objects (see page 2227) • Using Critical Sections (see page 2227) • Using a Multi-read Exclusive-write Synchronizer (see page 2228)
Coordinating Threads (see page 2227)	When writing the code that runs when your thread is executed, you must consider the behavior of other threads that may be executing simultaneously. In particular, care must be taken to avoid two threads trying to use the same global object or variable at the same time. In addition, the code in one thread can depend on the results of tasks performed by other threads. Whether using thread objects or generating threads using *BeginThread*, the following topics describe techniques for coordinating threads: • Avoiding Simultaneous Access (see page 2226) • Waiting for Other Threads (see page 2230) • Using the Main VCL Thread (see page 2235) When global memory does not... more (see page 2227)
Locking Objects (see page 2227)	Some objects have built-in locking that prevents the execution of other threads from using that object instance. For example, canvas objects (*TCanvas* and descendants) have a Lock method that prevents other threads from accessing the canvas until the Unlock method is called. VCL applications also include a thread-safe list object, TThreadList. Calling LockList returns the list object while also blocking other execution threads from using the list until the UnlockList method is called. Calls to *TCanvas.Lock* or *TThreadList.LockList* can be safely nested. The lock is not released until the last locking call is matched with a corresponding unlock call... more (see page 2227)
Using Critical Sections (see page 2227)	If objects do not provide built-in locking, you can use a critical section. Critical sections work like gates that allow only a single thread to enter at a time. To use a critical section, create a global instance of TCriticalSection. *TCriticalSection* has two methods, Acquire*(which blocks other threads from executing the section)* and Release*(which removes the block).* *Each critical section is associated with the global memory you want to protect. Every thread that accesses that global memory should first use the Acquire method to ensure that no other thread is using it. When finished, threads call the Release... more (see page 2227)*

Using the Multi-read Exclusive-write Synchronizer (see page 2228)	When you use critical sections to protect global memory, only one thread can use the memory at a time. This can be more protection than you need, especially if you have an object or variable that must be read often but to which you very seldom write. There is no danger in multiple threads reading the same memory simultaneously, as long as no thread is writing to it. When you have some global memory that is read often, but to which threads occasionally write, you can protect it using TMultiReadExclusiveWriteSynchronizer. This object acts like a critical section, but allows multiple... more (see page 2228)
Waiting for a Task to Be Completed (see page 2229)	Sometimes, you need to wait for a thread to finish some operation rather than waiting for a particular thread to complete execution. To do this, use an event object. Event objects (TEvent) should be created with global scope so that they can act like signals that are visible to all threads. When a thread completes an operation that other threads depend on, it calls *TEvent.SetEvent*. SetEvent turns on the signal, so any other thread that checks will know that the operation has completed. To turn off the signal, use the ResetEvent method. For example, consider a situation where you... more (see page 2229)
Waiting for a Thread to Finish Executing (see page 2230)	To wait for another thread to finish executing, use the WaitFor method of that other thread. *WaitFor* doesn't return until the other thread terminates, either by finishing its own Execute method or by terminating due to an exception. For example, the following code waits until another thread fills a thread list object before accessing the objects in the list:
Waiting for Other Threads (see page 2230)	If your thread must wait for another thread to finish some task, you can tell your thread to temporarily suspend execution. You can either • Wait for another thread to completely finish executing (see page 2230), or • Wait for a task to be completed (see page 2229).

3.2.4.15.1.1 Other Techniques for Sharing Memory

When using VCL objects, use the main thread to execute your code. Using the main thread ensures that the object does not indirectly access any memory that is also used by VCL objects in other threads. See Using the Main VCL Thread (see page 2235) for more information on the main thread.

If the global memory does not need to be shared by multiple threads, consider using thread-local variables instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads. See Using Thread-local Variables (see page 2237) for more information about thread-local variables.

See Also

Locking Objects (see page 2227)

Using the Main VCL Thread (see page 2235)

Using Critical Sections (see page 2227)

Using the Multi-read Exclusive-write Synchronizer (see page 2228)

3.2.4.15.1.2 Avoiding Simultaneous Access

To avoid clashing with other threads when accessing global objects or variables, you may need to block the execution of other threads until your thread code has finished an operation. Be careful not to block other execution threads unnecessarily. Doing so can cause performance to degrade seriously and negate most of the advantages of using multiple threads.

Three techniques prevent other threads from accessing the same memory as your thread:

- Locking Objects (see page 2227)
- Using Critical Sections (see page 2227)
- Using a Multi-read Exclusive-write Synchronizer (see page 2228)

See Also

Using the Main VCL Thread (see page 2235)

Using Thread-local Variables (see page 2237)

Waiting for Other Threads (see page 2230)

3.2.4.15.1.3 Coordinating Threads

When writing the code that runs when your thread is executed, you must consider the behavior of other threads that may be executing simultaneously. In particular, care must be taken to avoid two threads trying to use the same global object or variable at the same time. In addition, the code in one thread can depend on the results of tasks performed by other threads.

Whether using thread objects or generating threads using *BeginThread*, the following topics describe techniques for coordinating threads:

- Avoiding Simultaneous Access (see page 2226)
- Waiting for Other Threads (see page 2230)
- Using the Main VCL Thread (see page 2235)

When global memory does not need to be shared by multiple threads, consider using thread-local variables (see page 2237) instead of global variables. By using thread-local variables, your thread does not need to wait for or lock out any other threads.

See Also

Executing Thread Objects (see page 2242)

Defining Thread Objects (see page 2233)

3.2.4.15.1.4 Locking Objects

Some objects have built-in locking that prevents the execution of other threads from using that object instance.

For example, canvas objects (*TCanvas* and descendants) have a Lock method that prevents other threads from accessing the canvas until the Unlock method is called.

VCL applications also include a thread-safe list object, TThreadList. Calling LockList returns the list object while also blocking other execution threads from using the list until the UnlockList method is called. Calls to *TCanvas.Lock* or *TThreadList.LockList* can be safely nested. The lock is not released until the last locking call is matched with a corresponding unlock call in the same thread.

See Also

Using Critical Sections (see page 2227)

Using the Main VCL Thread (see page 2235)

Using the Multi-read Exclusive-write Synchronizer (see page 2228)

3.2.4.15.1.5 Using Critical Sections

If objects do not provide built-in locking, you can use a critical section. Critical sections work like gates that allow only a single thread to enter at a time. To use a critical section, create a global instance of TCriticalSection. *TCriticalSection* has two methods, Acquire*(which blocks other threads from executing the section) and Release(which removes the block).*

Each critical section is associated with the global memory you want to protect. Every thread that accesses that global memory should first use the Acquire method to ensure that no other thread is using it. When finished, threads call the Release method so that other threads can access the global memory by calling Acquire.

Warning: *Critical sections only work if every thread uses them to access the associated global memory. Threads that ignore the critical section and access the global memory without calling Acquire can introduce problems of simultaneous access.*

For example, consider an application that has a global critical section variable, LockXY, that blocks access to global variables X and Y. Any thread that uses X or Y must surround that use with calls to the critical section such as the following:

```
LockXY.Acquire; { lock out other threads }
try
Y := sin(X);
finally
LockXY.Release;
end;
```

```
pLockXY->Acquire(); // lock out other threads
try
{
Y = sin(X);
}
__finally
{
pLockXY->Release();
}
```

See Also

Locking Objects (see page 2227)

Using the Main VCL Thread (see page 2235)

Using the Multi-read Exclusive-write Synchronizer (see page 2228)

3.2.4.15.1.6 Using the Multi-read Exclusive-write Synchronizer

When you use critical sections to protect global memory, only one thread can use the memory at a time. This can be more protection than you need, especially if you have an object or variable that must be read often but to which you very seldom write. There is no danger in multiple threads reading the same memory simultaneously, as long as no thread is writing to it.

When you have some global memory that is read often, but to which threads occasionally write, you can protect it using TMultiReadExclusiveWriteSynchronizer. This object acts like a critical section, but allows multiple threads to read the memory it protects as long as no thread is writing to it. Threads must have exclusive access to write to memory protected by *TMultiReadExclusiveWriteSynchronizer*.

To use a multi-read exclusive-write synchronizer, create a global instance of *TMultiReadExclusiveWriteSynchronizer* that is associated with the global memory you want to protect. Every thread that reads from this memory must first call the BeginRead method. *BeginRead* ensures that no other thread is currently writing to the memory. When a thread finishes reading the protected memory, it calls the EndRead method. Any thread that writes to the protected memory must call BeginWrite first. *BeginWrite* ensures that no other thread is currently reading or writing to the memory. When a thread finishes writing to the protected memory, it calls the EndWrite method, so that threads waiting to read the memory can begin.

Warning: Like critical sections, the multi-read exclusive-write synchronizer only works if every thread uses it to access the associated global memory. Threads that ignore the synchronizer and access the global memory without calling *BeginRead* or *BeginWrite* introduce problems of simultaneous access.

See Also

Locking Objects (see page 2227)

Using the Main VCL Thread (see page 2235)

Using Critical Sections (see page 2227)

3.2.4.15.1.7 Waiting for a Task to Be Completed

Sometimes, you need to wait for a thread to finish some operation rather than waiting for a particular thread to complete execution. To do this, use an event object. Event objects (TEvent) should be created with global scope so that they can act like signals that are visible to all threads.

When a thread completes an operation that other threads depend on, it calls *TEvent.SetEvent*. SetEvent turns on the signal, so any other thread that checks will know that the operation has completed. To turn off the signal, use the ResetEvent method.

For example, consider a situation where you must wait for several threads to complete their execution rather than a single thread. Because you don't know which thread will finish last, you can't simply use the WaitFor method of one of the threads. Instead, you can have each thread increment a counter when it is finished, and have the last thread signal that they are all done by setting an event.

The following code shows the end of the OnTerminate event handler for all of the threads that must complete. *CounterGuard* is a global critical section object that prevents multiple threads from using the counter at the same time. *Counter* is a global variable that counts the number of threads that have completed.

```
procedure TDataModule.TaskThreadTerminate(Sender: TObject);
begin
...
CounterGuard.Acquire; { obtain a lock on the counter }
Dec(Counter);   { decrement the global counter variable }
if Counter = 0 then
Event1.SetEvent; { signal if this is the last thread }
CounterGuard.Release; { release the lock on the counter }
...
end;
void __fastcall TDataModule::TaskThreadTerminate(TObject *Sender)
{
...
CounterGuard->Acquire(); // lock the counter
if (--Counter == 0)    // decrement the global counter
Event1->SetEvent(); // signal if this is the last thread
CounterGuard->Release(); // release the lock on the counter
}
```

The main thread initializes the Counter variable, launches the task threads, and waits for the signal that they are all done by calling the WaitFor method. *WaitFor* waits for a specified time period for the signal to be set, and returns one of the values from the following table:

WaitFor return values

Value	Meaning
wrSignaled	The signal of the event was set.
wrTimeout	The specified time elapsed without the signal being set.
wrAbandoned	The event object was destroyed before the time-out period elapsed.
wrError	An error occurred while waiting.

The following shows how the main thread launches the task threads and then resumes when they have all completed:

```
Event1.ResetEvent; { clear the event before launching the threads }
for i := 1 to Counter do
TaskThread.Create(False); { create and launch task threads }
if Event1.WaitFor(20000) <> wrSignaled then
raise Exception;
{ now continue with the main thread. All task threads have finished }
Event1->ResetEvent(); // clear the event before launching the threads
for (int i = 0; i < Counter; i++)
```

```
new TaskThread(false); // create and launch task threads
if (Event1->WaitFor(20000) != wrSignaled)
throw Exception;
// now continue with the main thread, all task threads have finished
```

Note: If you do not want to stop waiting for an event after a specified time period, pass the *WaitFor* method a parameter value of INFINITE. Be careful when using INFINITE, because your thread will hang if the anticipated signal is never received.

See Also

Waiting for a Thread to Finish Executing (see page 2230)

Using the Main VCL Thread (see page 2235)

3.2.4.15.1.8 Waiting for a Thread to Finish Executing

To wait for another thread to finish executing, use the WaitFor method of that other thread. *WaitFor* doesn't return until the other thread terminates, either by finishing its own Execute method or by terminating due to an exception. For example, the following code waits until another thread fills a thread list object before accessing the objects in the list:

```
if ListFillingThread.WaitFor then
begin
with ThreadList1.LockList do
begin
for I := 0 to Count - 1 do
ProcessItem(Items[I]);
end;
ThreadList1.UnlockList;
end;
if (pListFillingThread->WaitFor())
{
  TList *pList = ThreadList1->LockList();
for (int i = 0; i < pList->Count; i++)
ProcessItem(pList->Items[i]);
ThreadList1->UnlockList();
}
```

In the previous example, the list items were only accessed when the *WaitFor* method indicated that the list was successfully filled. This return value must be assigned by the *Execute* method of the thread that was waited for. However, because threads that call *WaitFor* want to know the result of thread execution, not code that calls *Execute*, the *Execute* method does not return any value. Instead, the *Execute* method sets the ReturnValue property. *ReturnValue* is then returned by the *WaitFor* method when it is called by other threads. Return values are integers. Your application determines their meaning.

See Also

Waiting for a Task to Be Completed (see page 2229)

3.2.4.15.1.9 Waiting for Other Threads

If your thread must wait for another thread to finish some task, you can tell your thread to temporarily suspend execution. You can either

- Wait for another thread to completely finish executing (see page 2230), or
- Wait for a task to be completed (see page 2229).

See Also

Using the Main VCL Thread (see page 2235)

Using Thread-local Variables (see page 2237)

3.2.4.15.2 Writing Multi-threaded Applications

Several objects make writing multi-threaded applications easier. Multi-threaded applications are applications that include several simultaneous paths of execution. While using multiple threads requires careful thought, it can enhance your programs by:

- **Avoiding bottlenecks.** With only one thread, a program must stop all execution when waiting for slow processes such as accessing files on disk, communicating with other machines, or displaying multimedia content. The CPU sits idle until the process completes. With multiple threads, your application can continue execution in separate threads while one thread waits for the results of a slow process.
- **Organizing program behavior.** Often, a program's behavior can be organized into several parallel processes that function independently. Use threads to launch a single section of code simultaneously for each of these parallel cases. Use threads to assign priorities to various program tasks so that you can give more CPU time to more critical tasks.
- **Multiprocessing.** If the system running your program has multiple processors, you can improve performance by dividing the work into several threads and letting them run simultaneously on separate processors.

  Note: Not all operating systems implement true multi-processing, even when it is supported by the underlying hardware. For example, Windows 9x only simulates multiprocessing, even if the underlying hardware supports it.

  The following topics discuss support for threads:

- Defining Thread Objects (see page 2233)
- Coordinating Threads (see page 2227)
- Executing Thread Objects (see page 2242)
- Debugging Multi-threaded Applications (see page 2241)

See Also

Blocking Connections (see page 2340)

Managing Multiple Sessions (see page 1659)

3.2.4.15.3 Defining thread objects

Topics

Name	Description
Checking for Termination by Other Threads (see page 2232)	Your thread object begins running when the *Execute* method is called (see Executing thread objects (see page 2242)) and continues until *Execute* finishes. This reflects the model that the thread performs a specific task, and then stops when it is finished. Sometimes, however, an application needs a thread to execute until some external criterion is satisfied. You can allow other threads to signal that it is time for your thread to finish executing by checking the Terminated property. When another thread tries to terminate your thread, it calls the Terminate method. *Terminate* sets your thread's *Terminated* property to *True*. It is... more (see page 2232)
Defining Thread Objects (see page 2233)	For most applications, you can use a thread object to represent an execution thread in your application. Thread objects simplify writing multi-threaded applications by encapsulating the most commonly needed uses of threads. **Note:** Thread objects do not allow you to control the security attributes or stack size of your threads. If you need to control these, you must use the BeginThread function. Even when using *BeginThread*, you can still benefit from some of the thread synchronization objects and methods described in Coordinating Threads (see page 2227).
Handling Exceptions in the Thread Function (see page 2234)	The *Execute* method must catch all exceptions that occur in the thread. If you fail to catch an exception in your thread function, your application can cause access violations. This may not be obvious when you are developing your application, because the IDE catches the exception, but when you run your application outside of the debugger, the exception will cause a runtime error and the application will stop running. To catch the exceptions that occur inside your thread function, add a **try...except** block to the implementation of the *Execute* method:

Initializing the Thread (see page 2234)	If you want to write initialization code for your new thread class, you must override the *Create* method. Add a new constructor to the declaration of your thread class and write the initialization code as its implementation. This is where you can assign a default priority for your thread and indicate whether it should be freed automatically when it finishes executing.
Using the Main VCL Thread (see page 2235)	When you use objects from the class hierarchy, their properties and methods are not guaranteed to be thread-safe. That is, accessing properties or executing methods may perform some actions that use memory which is not protected from the actions of other threads. Because of this, a main thread is set aside to access VCL objects. This is the thread that handles all Windows messages received by components in your application. If all objects access their properties and execute their methods within this single thread, you need not worry about your objects interfering with each other. To use the main thread,... more (see page 2235)
Using Thread-local Variables (see page 2237)	The thread function and any of the routines it calls have their own local variables, just like any other Dephi language routines. These routines also can access any global variables. In fact, global variables provide a powerful mechanism for communicating between threads. Sometimes, however, you may want to use variables that are global to all the routines running in your thread, but not shared with other instances of the same thread class. You can do this by declaring thread-local variables. Make a variable thread-local by declaring it in a **threadvar** section (Delphi) or adding the **__thread** modifier (C++). For example,... more (see page 2237)
Writing Clean-up Code (see page 2237)	You can centralize the code that cleans up when your thread finishes executing. Just before a thread shuts down, an OnTerminate event occurs. Put any clean-up code in the *OnTerminate* event handler to ensure that it is always executed, no matter what execution path the Execute method follows. The *OnTerminate* event handler is not run as part of your thread. Instead, it is run in the context of the main VCL thread (see page 2235) of your application. This has two implications: • You can't use any thread-local variables (see page 2237) in an *OnTerminate* event handler (unless you want the main VCL thread values). • You can... more (see page 2237)
Writing the Thread Function (see page 2238)	The Execute method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program because you must make sure that you don't overwrite memory that is used by other threads in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads. When implementing the *Execute* method, you can manage these issues by: • Using thread-local variables (see page 2237)... more (see page 2238)

3.2.4.15.3.1 Checking for Termination by Other Threads

Your thread object begins running when the *Execute* method is called (see Executing thread objects (see page 2242)) and continues until *Execute* finishes. This reflects the model that the thread performs a specific task, and then stops when it is finished. Sometimes, however, an application needs a thread to execute until some external criterion is satisfied.

You can allow other threads to signal that it is time for your thread to finish executing by checking the Terminated property. When another thread tries to terminate your thread, it calls the Terminate method. *Terminate* sets your thread's *Terminated* property to *True*. It is up to your Execute method to implement the *Terminate* method by checking and responding to the *Terminated* property. The following example shows one way to do this:

```
procedure TMyThread.Execute;
begin
while not Terminated do
PerformSomeTask;
end;
void __fastcall TMyThread::Execute()
{
while (!Terminated)
```

```
PerformSomeTask();
}
```

See Also

Using the Main VCL Thread (see page 2235)

Waiting for Other Threads (see page 2230)

Using Thread-local Variables (see page 2237)

Handling Exceptions in the Thread Function (see page 2234)

3.2.4.15.3.2 Defining Thread Objects

For most applications, you can use a thread object to represent an execution thread in your application. Thread objects simplify writing multi-threaded applications by encapsulating the most commonly needed uses of threads.

Note: Thread objects do not allow you to control the security attributes or stack size of your threads. If you need to control these, you must use the BeginThread function. Even when using *BeginThread*, you can still benefit from some of the thread synchronization objects and methods described in Coordinating Threads (see page 2227).

To use a thread object in your application

1. Create a new descendant of TThread, choose **File** ▶ **New** ▶ **Other** from the main menu.
2. In the New Items dialog box under Delphi Files, double-click Thread Object and enter a class name, such as *TMyThread*.
3. Check the Named Thread check box and enter a thread name (VCL applications only).
4. Click OK, the **Code Editor** creates a new unit file to implement the thread.

For more information on naming threads, see Naming a Thread (see page 2241).

Note: Unlike most dialog boxes in the IDE that require a class name, the New Thread Object dialog box does not automatically prepend a 'T' to the front of the class name you provide.

The automatically generated unit file contains the skeleton code for your new thread class. If you named your thread *TMyThread*, it would look like the following:

```
unit Unit2;
interface
uses
Classes;
type
TMyThread = class(TThread)
private
{ Private declarations }
protected
procedure Execute; override;
end;
implementation
{ TMyThread }
procedure TMyThread.Execute;
begin
{ Place thread code here }
end;
end.
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit2.h"
#pragma package(smart_init)
//---------------------------------------------------------------------------
__fastcall TMyThread::TMyThread(bool CreateSuspended): TThread(CreateSuspended)
{
}
```

```
//---------------------------------------------------------------------------
void __fastcall TMyThread::Execute()
{
// ---- Place thread code here ----
}
//---------------------------------------------------------------------------
```

In the automatically generated unit file, you

- Optionally, initialize the thread (see page 2234).
- Write the thread function (see page 2238) by filling in the *Execute* method.
- Optionally, write clean-up code (see page 2237).

See Also

Executing Thread Objects (see page 2242)

3.2.4.15.3.3 Handling Exceptions in the Thread Function

The *Execute* method must catch all exceptions that occur in the thread. If you fail to catch an exception in your thread function, your application can cause access violations. This may not be obvious when you are developing your application, because the IDE catches the exception, but when you run your application outside of the debugger, the exception will cause a runtime error and the application will stop running.

To catch the exceptions that occur inside your thread function, add a **try...except** block to the implementation of the *Execute* method:

```
procedure TMyThread.Execute;
begin
  try
    while not Terminated do
  PerformSomeTask;
  except
    { do something with exceptions }
  end;
end;
void __fastcall TMyThread::Execute()
{
  try
  {
    while (!Terminated)
      PerformSomeTask();
  }
  catch (...)
  {
    // do something with exceptions
  }
}
```

See Also

Using the Main VCL Thread (see page 2235)

Waiting for Other Threads (see page 2230)

Using Thread-local Variables (see page 2237)

Checking for Termination by Other Threads (see page 2232)

3.2.4.15.3.4 Initializing the Thread

If you want to write initialization code for your new thread class, you must override the *Create* method. Add a new constructor to the declaration of your thread class and write the initialization code as its implementation. This is where you can assign a default

priority for your thread and indicate whether it should be freed automatically when it finishes executing.

Assigning a default priority

Priority indicates how much preference the thread gets when the operating system schedules CPU time among all the threads in your application. Use a high priority thread to handle time critical tasks, and a low priority thread to perform other tasks. To indicate the priority of your thread object, set the Priority property.

If writing a Windows-only application, *Priority* values fall along a scale, as described in the following table:

Thread priorities

Value	Priority
tpIdle	The thread executes only when the system is idle. Windows won't interrupt other threads to execute a thread with *tpIdle* priority.
tpLowest	The thread's priority is two points below normal.
tpLower	The thread's priority is one point below normal.
tpNormal	The thread has normal priority.
tpHigher	The thread's priority is one point above normal.
tpHighest	The thread's priority is two points above normal.
tpTimeCritical	The thread gets highest priority.

Warning: Boosting the thread priority of a CPU intensive operation may "starve" other threads in the application. Only apply priority boosts to threads that spend most of their time waiting for external events.

The following code shows the constructor of a low-priority thread that performs background tasks which should not interfere with the rest of the application's performance:

```
constructor TMyThread.Create(CreateSuspended: Boolean);
begin
  inherited Create(CreateSuspended);
Priority := tpIdle;
end;
//---------------------------------------------------------------------------
__fastcall TMyThread::TMyThread(bool CreateSuspended): TThread(CreateSuspended)
{
Priority = tpIdle;
}
//---------------------------------------------------------------------------
```

Indicating when threads are freed

Usually, when threads finish their operation, they can simply be freed. In this case, it is easiest to let the thread object free itself. To do this, set the FreeOnTerminate property to *True*.

There are times, however, when the termination of a thread must be coordinated with other threads. For example, you may be waiting for one thread to return a value before performing an action in another thread. To do this, you do not want to free the first thread until the second has received the return value. You can handle this situation by setting *FreeOnTerminate* to *False* and then explicitly freeing the first thread from the second.

See Also

Writing the Thread Function (see page 2238)

Writing Clean-up Code (see page 2237)

3.2.4.15.3.5 Using the Main VCL Thread

When you use objects from the class hierarchy, their properties and methods are not guaranteed to be thread-safe. That is,

accessing properties or executing methods may perform some actions that use memory which is not protected from the actions of other threads. Because of this, a main thread is set aside to access VCL objects. This is the thread that handles all Windows messages received by components in your application.

If all objects access their properties and execute their methods within this single thread, you need not worry about your objects interfering with each other. To use the main thread, create a separate routine that performs the required actions. Call this separate routine from within your thread's Synchronize method. For example:

```
procedure TMyThread.PushTheButton;
begin
Button1.Click;
end;
...
procedure TMyThread.Execute;
begin
...
Synchronize(PushTheButton);
...
end;
void __fastcall TMyThread::PushTheButton(void)
{
Button1->Click();
}
void __fastcall TMyThread::Execute()
{
...
Synchronize((TThreadMethod)PushTheButton);
...
}
```

Synchronize waits for the main thread to enter the message loop and then executes the passed method.

Note: Because *Synchronize* uses the message loop, it does not work in console applications. You must use other mechanisms, such as critical sections, to protect access to VCL objects in console applications.

You do not always need to use the main thread. Some objects are thread-aware. Omitting the use of the *Synchronize* method when you know an object's methods are thread-safe will improve performance because you don't need to wait for the VCL thread to enter its message loop. You do not need to use the *Synchronize* method for the following objects:

Object	Description
Data access component	Data access components are thread-safe as follows: For BDE-enabled datasets, each thread must have its own database session component. The one exception to this is when you are using Microsoft Access drivers, which are built using a Microsoft library that is not thread-safe. For dbExpress, as long as the vendor client library is thread-safe, the dbExpress components will be thread-safe. ADO and InterBaseExpress components are thread-safe. When using data access components, you must still wrap all calls that involve data-aware controls in the *Synchronize* method. Thus, for example, you need to synchronize calls that link a data control to a dataset by setting the *DataSet* property of the data source object, but you don't need to synchronize to access the data in a field of the dataset. For more information about using database sessions with threads in BDE-enabled applications, see Managing multiple sessions (see page 1659).
Control	Controls are not thread-safe.
Graphic	Graphics objects are thread-safe. You do not need to use the main VCL thread to access TFont, TPen, TBrush, TBitmap, TMetafile (VCL only), or TTIcon. Canvas objects can be used outside the *Synchronize* method by locking them (see page 2227).
List	While list objects are not thread-safe, you can use a thread-safe version, TThreadList, instead of *TList*.

Call the *CheckSynchronize* routine periodically within the main thread of your application so that background threads can synchronize their execution with the main thread. The best place to call *CheckSynchronize* is when the application is idle (for

example, from an *OnIdle* event handler). This ensures that it is safe to make method calls in the background thread.

See Also

Using Thread-local Variables (see page 2237)

Waiting for Other Threads (see page 2230)

Checking for Termination by Other Threads (see page 2232)

Handling Exceptions in the Thread Function (see page 2234)

3.2.4.15.3.6 Using Thread-local Variables

The thread function and any of the routines it calls have their own local variables, just like any other Dephi language routines. These routines also can access any global variables. In fact, global variables provide a powerful mechanism for communicating between threads.

Sometimes, however, you may want to use variables that are global to all the routines running in your thread, but not shared with other instances of the same thread class. You can do this by declaring thread-local variables. Make a variable thread-local by declaring it in a **threadvar** section (Delphi) or adding the **__thread** modifier (C++). For example,

```
threadvar
x : integer;
```

```
int __thread x;
```

declares an integer type variable that is private to each thread in the application, but global within each thread.

The threadvar section can only be used for global variables. Pointer and Function variables can't be thread variables. Types that use copy-on-write semantics, such as long strings don't work as thread variables either.

In C++, the following declarations require runtime initialization and are therefore illegal:

```
int f( );
int __thread x = f( );   // illegal
```

Instantiation of a class with a user-defined constructor or destructor requires runtime initialization and is therefore illegal:

```
class X {
    X( );
    ~X( );
};
X __thread myclass;         // illegal
```

See Also

Using the Main VCL Thread (see page 2235)

Waiting for Other Threads (see page 2230)

Checking for Termination by Other Threads (see page 2232)

Handling Exceptions in the Thread Function (see page 2234)

3.2.4.15.3.7 Writing Clean-up Code

You can centralize the code that cleans up when your thread finishes executing. Just before a thread shuts down, an OnTerminate event occurs. Put any clean-up code in the *OnTerminate* event handler to ensure that it is always executed, no matter what execution path the Execute method follows.

The *OnTerminate* event handler is not run as part of your thread. Instead, it is run in the context of the main VCL thread (see page 2235) of your application. This has two implications:

- You can't use any thread-local variables (see page 2237) in an *OnTerminate* event handler (unless you want the main VCL

thread values).

- You can safely access any objects from the *OnTerminate* event handler without worrying about clashing with other threads.

See Also

Initializing the Thread (see page 2234)

Writing the Thread Function (see page 2238)

3.2.4.15.3.8 Writing the Thread Function

The Execute method is your thread function. You can think of it as a program that is launched by your application, except that it shares the same process space. Writing the thread function is a little trickier than writing a separate program because you must make sure that you don't overwrite memory that is used by other threads in your application. On the other hand, because the thread shares the same process space with other threads, you can use the shared memory to communicate between threads.

When implementing the *Execute* method, you can manage these issues by:

- Using thread-local variables (see page 2237)
- Avoiding simultaneous access (see page 2226)
- Waiting for other threads (see page 2230)
- Checking for termination by other threads (see page 2232)
- Handling exceptions in the thread function (see page 2234)

See Also

Initializing the Thread (see page 2234)

Writing Clean-up Code (see page 2237)

3.2.4.15.4 Executing thread objects

Topics

Name	Description
Assigning Separate Names to Similar Threads (see page 2239)	All thread instances from the same thread class have the same name. However, you can assign a different name for each thread instance at runtime using the following steps.
Converting an Unnamed Thread to a Named Thread (see page 2239)	You can convert an unnamed thread to a named thread. For example, if you have a thread class that was created using Delphi 6 or earlier, convert it into a named thread.
Naming a Thread (see page 2241)	Because it is difficult to tell which thread ID refers to which thread in the Thread Status box, you can name your thread classes. When you are creating a thread class in the Thread Object dialog box, besides entering a class name, also check the Named Thread check box, enter a thread name, and click OK. Naming the thread class adds a method to your thread class called *SetName*. When the thread starts running, it calls the *SetName* method first. **Note:** You can name threads in VCL applications only. You can also: • Convert an unnamed thread to a named (see page 2239)... more (see page 2241)
Debugging Multi-threaded Applications (see page 2241)	When debugging multi-threaded applications, it can be confusing trying to keep track of the status of all the threads that are executing simultaneously, or even to determine which thread is executing when you stop at a breakpoint. You can use the Thread Status box to help you keep track of and manipulate all the threads in your application. To display the Thread status box, choose **View ▶ Debug Windows ▶ Threads** from the main menu. When a debug event occurs (breakpoint, exception, paused), the thread status view indicates the status of each thread. Right-click the Thread Status box to access commands that... more (see page 2241)

Executing Thread Objects (see page 2242)	Once you have implemented a thread class by giving it an *Execute* method, you can use it in your application to launch the code in the *Execute* method. To use a thread, first create an instance of the thread class. You can create a thread instance that starts running immediately, or you can create your thread in a suspended state so that it only begins when you call the Resume method. To create a thread so that it starts up immediately, set the constructor's *CreateSuspended* parameter to *False*. For example, the following line creates a thread and starts its... more (see page 2242)
Overriding the Default Priority (see page 2242)	When the amount of CPU time the thread should receive is implicit in the thread's task, its priority is set in the constructor. This is described in Initializing the thread (see page 2234). However, if the thread priority varies depending on when the thread is executed, create the thread in a suspended state, set the priority, and then start the thread running:
Starting and Stopping Threads (see page 2242)	A thread can be started and stopped any number of times before it finishes executing. To stop a thread temporarily, call its Suspend method. When it is safe for the thread to resume, call its Resume method. *Suspend* increases an internal counter, so you can nest calls to *Suspend* and *Resume*. The thread does not resume execution until all suspensions have been matched by a call to *Resume*. You can request that a thread end execution prematurely by calling the Terminate method. *Terminate* sets the thread's Terminated property to *True*. If you have implemented the *Execute* method... more (see page 2242)

3.2.4.15.4.1 Assigning Separate Names to Similar Threads

All thread instances from the same thread class have the same name. However, you can assign a different name for each thread instance at runtime using the following steps.

To assign separate names to similar threads

1. Add a *ThreadName* property to the thread class by adding the following in the class definition:

```
property ThreadName: string read FName write FName;
__property AnsiString ThreadName = {read=FName, write=FName};
```

2. In the *SetName* method, change where it says:

```
ThreadNameInfo.FName := 'MyThreadName';
info.szName = "MyThreadName";
```

to:

```
ThreadNameInfo.FName := ThreadName;
info.szName = ThreadName;
```

To create the thread object

1. Create it suspended. See Executing Thread Objects (see page 2242).
2. Assign a name, such as `MyThread.ThreadName := 'SearchForFiles';`
3. Resume the thread. See Starting and Stopping Threads (see page 2242).

See Also

Converting an Unnamed Thread to a Named Thread (see page 2239)

Naming a Thread (see page 2241)

3.2.4.15.4.2 Converting an Unnamed Thread to a Named Thread

You can convert an unnamed thread to a named thread. For example, if you have a thread class that was created using Delphi 6 or earlier, convert it into a named thread.

To convert an unnamed thread to a named thread

1. Add the Windows unit to the **uses** clause of the unit your thread is declared in:

```
//---------------------------------------------------------------------------
```

```
uses
Classes {$IFDEF MSWINDOWS} , Windows {$ENDIF};
//---------------------------------------------------------------------------
```

2. Add the *SetName* method to your thread class in the **interface** section:

```
//---------------------------------------------------------------------------
type
TMyThread = class(TThread)
private
procedure SetName;
protected
procedure Execute; override;
end;
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
void TMyThread::SetName()
{
THREADNAME_INFO info;
info.dwType = 0x1000;
info.szName = "MyThreadName";
info.dwThreadID = -1;
info.dwFlags = 0;
__try
{
RaiseException( 0x406D1388, 0, sizeof(info)/sizeof(DWORD),(DWORD*)&info );
}
__except (EXCEPTION_CONTINUE_EXECUTION)
{
}
}
//---------------------------------------------------------------------------
```

3. Add the *TThreadNameInfo* record and *SetName* method in the **implementation** section:

```
//---------------------------------------------------------------------------
{$IFDEF MSWINDOWS}
type
TThreadNameInfo = record
FType: LongWord;     // must be 0x1000
FName: PChar;        // pointer to name (in user address space)
FThreadID: LongWord; // thread ID (-1 indicates caller thread)
FFlags: LongWord;    // reserved for future use, must be zero
  end;
{$ENDIF}
{ TMyThread }
procedure TMyThread.SetName;
{$IFDEF MSWINDOWS}
var
ThreadNameInfo: TThreadNameInfo;
{$ENDIF}
begin
{$IFDEF MSWINDOWS}
ThreadNameInfo.FType := $1000;
ThreadNameInfo.FName := 'MyThreadName';
ThreadNameInfo.FThreadID := $FFFFFFFF;
ThreadNameInfo.FFlags := 0;
try
RaiseException( $406D1388, 0, sizeof(ThreadNameInfo) div sizeof(LongWord), @ThreadNameInfo );
 except
  end;
{$ENDIF}
end;
//---------------------------------------------------------------------------
```

Note: Set *TThreadNameInfo* to the name of your thread class.

The debugger sees the exception and looks up the thread name in the structure you pass in. When debugging, the debugger

displays the name of the thread in the Thread Status box's thread ID field.

4. Add a call to the new *SetName* method at the beginning of your thread's *Execute* method:

```
//---------------------------------------------------------------------------
procedure TMyThread.Execute;
begin
SetName;
{ Place thread code here }
end;
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
void __fastcall TMyThread::Execute()
{
SetName();
//---- Place existing Execute method code here ----
}
//---------------------------------------------------------------------------
```

See Also

Naming a Thread (see page 2241)

Assigning Separate Names to Similar Threads (see page 2239)

3.2.4.15.4.3 Naming a Thread

Because it is difficult to tell which thread ID refers to which thread in the Thread Status box, you can name your thread classes. When you are creating a thread class in the Thread Object dialog box, besides entering a class name, also check the Named Thread check box, enter a thread name, and click OK.

Naming the thread class adds a method to your thread class called *SetName*. When the thread starts running, it calls the *SetName* method first.

Note: You can name threads in VCL applications only.

You can also:

- Convert an unnamed thread to a named thread (see page 2239).
- Assign separate names to similar threads (see page 2239).

See Also

Executing Thread Objects (see page 2242)

Defining Thread Objects (see page 2233)

Converting an Unnamed Thread to a Named Thread (see page 2239)

3.2.4.15.4.4 Debugging Multi-threaded Applications

When debugging multi-threaded applications, it can be confusing trying to keep track of the status of all the threads that are executing simultaneously, or even to determine which thread is executing when you stop at a breakpoint. You can use the Thread Status box to help you keep track of and manipulate all the threads in your application. To display the Thread status box, choose **View** ▸ **Debug Windows** ▸ **Threads** from the main menu.

When a debug event occurs (breakpoint, exception, paused), the thread status view indicates the status of each thread. Right-click the Thread Status box to access commands that locate the corresponding source location or make a different thread current. When a thread is marked as current, the next step or run operation is relative to that thread.

The Thread Status box lists all your application's execution threads by their thread ID. If you are using thread objects, the thread ID is the value of the *ThreadID* property. If you are not using thread objects, the thread ID for each thread is returned by the call to *BeginThread*.

See Also

Executing Thread Objects (see page 2242)

Defining Thread Objects (see page 2233)

Defining Thread Objects (see page 2233)

Naming a Thread (see page 2241)

Starting and Stopping Threads (see page 2242)

3.2.4.15.4.5 Executing Thread Objects

Once you have implemented a thread class by giving it an *Execute* method, you can use it in your application to launch the code in the *Execute* method. To use a thread, first create an instance of the thread class. You can create a thread instance that starts running immediately, or you can create your thread in a suspended state so that it only begins when you call the Resume method. To create a thread so that it starts up immediately, set the constructor's *CreateSuspended* parameter to *False*. For example, the following line creates a thread and starts its execution:

```
SecondThread := TMyThread.Create(false); {create and run the thread }
TMyThread *SecondThread = new TMyThread(false); // create and run the thread
```

Warning: Do not create too many threads in your application. The overhead in managing multiple threads can impact performance. The recommended limit is 16 threads per process on single processor systems. This limit assumes that most of those threads are waiting for external events. If all threads are active, you will want to use fewer.

You can create multiple instances of the same thread type to execute parallel code. For example, you can launch a new instance of a thread in response to some user action, allowing each thread to perform the expected response.

The following topics discuss how to use the threads in your application:

- Overriding the Default Priority (see page 2242).
- Starting and Stopping Threads (see page 2242)

See Also

Defining Thread Objects (see page 2233)

3.2.4.15.4.6 Overriding the Default Priority

When the amount of CPU time the thread should receive is implicit in the thread's task, its priority is set in the constructor. This is described in Initializing the thread (see page 2234). However, if the thread priority varies depending on when the thread is executed, create the thread in a suspended state, set the priority, and then start the thread running:

```
SecondThread :=  TMyThread.Create(True); { create but don't run }
SecondThread.Priority := tpLower; { set the priority lower than normal }
SecondThread.Resume; { now run the thread }
TMyThread *SecondThread = new TMyThread(true); // create but don't run
SecondThread->Priority = tpLower; // set the priority lower than normal
SecondThread->Resume(); // now run the thread
```

See Also

Starting and Stopping Threads (see page 2242)

3.2.4.15.4.7 Starting and Stopping Threads

A thread can be started and stopped any number of times before it finishes executing. To stop a thread temporarily, call its Suspend method. When it is safe for the thread to resume, call its Resume method. *Suspend* increases an internal counter, so you can nest calls to *Suspend* and *Resume*. The thread does not resume execution until all suspensions have been matched by

a call to *Resume*.

You can request that a thread end execution prematurely by calling the Terminate method. *Terminate* sets the thread's Terminated property to *True*. If you have implemented the *Execute* method properly, it checks the *Terminated* property periodically, and stops execution when *Terminated* is *True*.

See Also

Overriding the Default Priority (see page 2242)

Checking for Termination by Other Threads (see page 2232)

3.2.5 Writing Internet Applications

Contains the Developer's Guide topics for writing internet applications in Delphi.

Topics

Name	Description
Creating Internet server applications (see page 2243)	
Using IntraWeb (VCL for the Web) (see page 2254)	For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.
Using Web Broker (see page 2261)	
Using Web Services (see page 2289)	
Using WebSnap (see page 2310)	WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.
Working with sockets (see page 2335)	
Working with XML documents (see page 2351)	

3.2.5.1 Creating Internet server applications

Topics

Name	Description
Debugging server applications (see page 2244)	
About Web Broker and WebSnap (see page 2246)	Part of the function of any application is to make data accessible to the user. In a standard application you accomplish this by creating traditional front end elements, like dialogs and scrolling windows. Developers can specify the exact layout of these objects using familiar form design tools. Web server applications must be designed differently, however. All information passed to users must be in the form of HTML pages which are transferred through HTTP. Pages are generally interpreted on the client machine by a Web browser application, which displays the pages in a form appropriate for the user's particular system in... more (see page 2246)
HTTP server activity (see page 2247)	
Creating Internet Applications: Overview (see page 2251)	Web server applications extend the functionality and capability of existing Web servers. A Web server application receives HTTP request messages from the Web server, performs any actions requested in those messages, and formulates responses that it passes back to the Web server. Many operations that you can perform with an ordinary application can be incorporated into a Web server application. The IDE provides two different architectures for developing Web server applications: Web Broker and WebSnap. Although these two architectures are different, WebSnap and Web Broker have many common elements. The WebSnap architecture acts as a superset of Web Broker. It... more (see page 2251)
Terminology and standards (see page 2252)	

3.2.5.1.1 Debugging server applications

Topics

Name	Description
Debugging Web Applications That Are DLLs (see page 2244)	ISAPI, NSAPI, and Apache applications are actually DLLs that contain predefined entry points. The Web server passes request messages to the application by making calls to these entry points. Because these applications are DLLs, you can debug them by setting your application's run parameters to launch the server. To set up your application's run parameters, choose **Run ▶ Parameters** and set the Host Application and Run Parameters to specify the executable for the Web server and any parameters it requires when you launch it. For details about these values on your Web server, see the documentation provided by you Web server... more (see page 2244)
Debugging Server Applications (see page 2245)	Debugging Web server applications presents some unique problems, because they run in response to messages from a Web server. You can not simply launch your application from the IDE, because that leaves the Web server out of the loop, and your application will not find the request message it is expecting. The following topics describe techniques you can use to debug Web server applications: • Using the Web Application Debugger (see page 2245) • Debugging Web Applications that are DLLs (see page 2244)
Using the Web Application Debugger (see page 2245)	The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the supported types of Web application and install it with a commercial Web server. To use the Web Application Debugger, you must first create your Web application as a Web Application Debugger executable. Whether you are using Web Broker or WebSnap, the wizard that creates your Web server application includes this as an option when you first begin the application.... more (see page 2245)

3.2.5.1.1.1 Debugging Web Applications That Are DLLs

ISAPI, NSAPI, and Apache applications are actually DLLs that contain predefined entry points. The Web server passes request messages to the application by making calls to these entry points. Because these applications are DLLs, you can debug them by setting your application's run parameters to launch the server.

To set up your application's run parameters, choose **Run ▶ Parameters** and set the Host Application and Run Parameters to specify the executable for the Web server and any parameters it requires when you launch it. For details about these values on your Web server, see the documentation provided by you Web server vendor.

Note: Some Web Servers require additional changes before you have the rights to launch the Host Application in this way. See your Web server vendor for details.

Tip: If you are using Windows 2000 with IIS 5, details on all of the changes you need to make to set up your rights properly are described at the following Web site:

`http://community.borland.com/article/0,1410,23024,00.html`

Once you have set the Host Application and Run Parameters, you can set up your breakpoints so that when the server passes a request message to your DLL, you hit one of your breakpoints, and can debug normally.

Note: Before launching the Web server using your application's run parameters, make sure that the server is not already running.

User rights necessary for DLL debugging

Under Windows, you must have the correct user rights to debug a DLL.

To obtain these rights

1. In the Administrative Tools portion of the Control Panel, click on Local Security Policy. Expand Local Policies and double-click User Rights Assignment. Double-click Act as part of the operating system in the right-hand panel.
2. Select Add to add a user to the list. Add your current user.
3. Reboot so the changes take effect.

See Also

HTTP Server Activity (see page 2248)

3.2.5.1.1.2 Debugging Server Applications

Debugging Web server applications presents some unique problems, because they run in response to messages from a Web server. You can not simply launch your application from the IDE, because that leaves the Web server out of the loop, and your application will not find the request message it is expecting.

The following topics describe techniques you can use to debug Web server applications:

- Using the Web Application Debugger (see page 2245)
- Debugging Web Applications that are DLLs (see page 2244)

See Also

Creating Web Server Applications (see page 2313)

The Structure of a Web Broker Application (see page 2286)

Types of Web Server Applications (see page 2249)

3.2.5.1.1.3 Using the Web Application Debugger

The Web Application Debugger provides an easy way to monitor HTTP requests, responses, and response times. The Web Application Debugger takes the place of the Web server. Once you have debugged your application, you can convert it to one of the supported types of Web application and install it with a commercial Web server.

To use the Web Application Debugger, you must first create your Web application as a Web Application Debugger executable. Whether you are using Web Broker or WebSnap, the wizard that creates your Web server application includes this as an option when you first begin the application. This creates a Web server application that is also a COM server.

For information on how to write this Web server application using Web Broker, see Using Web Broker (see page 2272). For more information on using WebSnap, see Creating Web Server applications using WebSnap (see page 2315).

Launching your application with the Web Application Debugger

Once you have developed your Web server application, you can run and debug it.

To launch your application with the Web Application Debugger

1. With your project loaded in the IDE, set any breakpoints so that you can debug your application just like any other executable.
2. Choose **Run** ▸ **Run**. This displays the console window of the COM server that is your Web server application. The first time you run your application, it registers your COM server so that the Web App debugger can access it.
3. Select **Tools** ▸ **Web App Debugger**.
4. Click the Start button. This displays the ServerInfo page in your default Browser.

5. The ServerInfo page provides a drop-down list of all registered Web Application Debugger executables. Select your application from the drop-down list. If you do not find your application in this drop-down list, try running your application as an executable. Your application must be run once so that it can register itself. If you still do not find your application in the drop-down list, try refreshing the Web page. (Sometimes the Web browser caches this page, preventing you from seeing the most recent changes.)
6. Once you have selected your application in the drop-down list, press the Go button. This launches your application in the Web Application Debugger, which provides you with details on request and response messages that pass between your application and the Web Application Debugger.

Converting your application to another type of Web server application

When you have finished debugging your Web server application with the Web Application Debugger, you will need to convert it to another type that can be installed on a commercial Web server. To learn more about converting your application, see "Converting Web server application target types" in the topic Types of Web server applications (see page 2249).

See Also

Debugging Web Applications That Are DLLs (see page 2244)

HTTP Server Activity (see page 2248)

3.2.5.1.2 About Web Broker and WebSnap

Part of the function of any application is to make data accessible to the user. In a standard application you accomplish this by creating traditional front end elements, like dialogs and scrolling windows. Developers can specify the exact layout of these objects using familiar form design tools. Web server applications must be designed differently, however. All information passed to users must be in the form of HTML pages which are transferred through HTTP. Pages are generally interpreted on the client machine by a Web browser application, which displays the pages in a form appropriate for the user's particular system in its present state.

The first step in building a Web server application is choosing which architecture you want to use, Web Broker or WebSnap. Both approaches provide many of the same features, including

- Support for CGI and Apache DSO Web server application types. These are described in Types of Web Server Applications (see page 2249).
- Multithreading support so that incoming client requests are handled on separate threads.
- Caching of Web modules for quicker responses.

Both the Web Broker and WebSnap components handle all of the mechanics of page transfer. WebSnap uses Web Broker as its foundation, so it incorporates all of the functionality of Web Broker's architecture. WebSnap offers a much more powerful set of tools for generating pages, however. Also, WebSnap applications allow you to use server-side scripting to help generate pages at runtime. Web Broker does not have this scripting capability. The tools offered in Web Broker are not nearly as complete as those in WebSnap, and are much less intuitive. If you are developing a new Web server application, WebSnap is probably a better choice of architecture than Web Broker.

The major differences between these two approaches are outlined in the following table:

Web Broker versus WebSnap

Web Broker	WebSnap
Backward compatible	Although WebSnap applications can use any Web Broker components that produce content, the Web modules and dispatcher that contain these are new.
Only one Web module allowed in an application.	Multiple Web modules can partition the application into units, allowing multiple developers to work on the same project with fewer conflicts.
Only one Web dispatcher allowed in the application.	Multiple, special-purpose dispatchers handle different types of requests.

Specialized components for creating content include page producers, InternetExpress components, and Web Services components.	Supports all the content producers that can appear in Web Broker applications, plus many others designed to let you quickly build complex data-driven Web pages.
No scripting support.	Support for server-side scripting allows HTML generation logic to be separated from the business logic.
No built-in support for named pages.	Named pages can be automatically retrieved by a page dispatcher and addressed from server-side scripts.
No session support.	Sessions store information about an end user that is needed for a short period of time. This can be used for such tasks as login/logout support.
Every request must be explicitly handled, using either an action item or an auto-dispatching component.	Dispatch components automatically respond to a variety of requests.
Only a few specialized components provide previews of the content they produce. Most development is not visual.	WebSnaplets you build Web pages more visually and view the results at design time. Previews are available for all components.

For more information on Web Broker, see Using Web Broker. (see page 2272) For more information on WebSnap, see Creating Web Server Applications Using WebSnap (see page 2315).

See Also

Creating Internet Applications: Overview (see page 2251)

3.2.5.1.3 HTTP server activity

Topics

Name	Description
Composing Client Requests (see page 2248)	When an HTML hypertext link is selected (or the user otherwise specifies a URL), the browser collects information about the protocol, the specified domain, the path to the information, the date and time, the operating environment, the browser itself, and other content information. It then composes a request. For example, to display a page of images based on criteria selected by clicking buttons on a form, the client might construct this URL:
HTTP Server Activity (see page 2248)	The client/server nature of Web browsers is deceptively simple. To most users, retrieving information on the World Wide Web is a simple procedure: click on a link, and the information appears on the screen. More knowledgeable users have some understanding of the nature of HTML syntax and the client/server nature of the protocols used. This is usually sufficient for the production of simple, page-oriented Web site content. Authors of more complex Web pages have a wide variety of options to automate the collection and presentation of information using HTML. Before building a Web server application, it is useful to understand... more (see page 2248)
Responding to Client Requests (see page 2249)	When a Web server application finishes with a client request, it constructs a page of HTML code or other MIME content, and passes it back (via the server) to the client for display. The way the response is sent may differ based on the type of program. When a DLL finishes, it passes the HTML page and any response information directly back to the server, which passes them back to the client. Creating a Web server application as a DLL reduces system load and resource use by reducing the number of processes and disk accesses necessary to service an individual... more (see page 2249)

Serving Client Requests (see page 2249)	The Web server receives a client request and can perform any number of actions, based on its configuration. If the server is configured to recognize the /gallery.dll portion of the request as a program, it passes information about the request to that program. The way information about the request is passed to the program depends on the type of Web server application: • If the program is a Common Gateway Interface (CGI) program, the server passes the information contained in the request directly to the CGI program. The server waits while the program executes. When the CGI program exits, it passes... more (see page 2249)
Types of Web Server Applications (see page 2249)	Whether you use Web Broker or WebSnap, you can create five standard types of Web server applications. In addition, you can create a Web Application Debugger executable, which integrates the Web server into your application so that you can debug your application logic. The Web Application Debugger executable is intended only for debugging. When you deploy your application, you should migrate to one of the other five types.

3.2.5.1.3.1 Composing Client Requests

When an HTML hypertext link is selected (or the user otherwise specifies a URL), the browser collects information about the protocol, the specified domain, the path to the information, the date and time, the operating environment, the browser itself, and other content information. It then composes a request.

For example, to display a page of images based on criteria selected by clicking buttons on a form, the client might construct this URL:

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

which specifies an HTTP server in the www.TSite.com domain. The client contacts www.TSite.com, connects to the HTTP server, and passes it a request. The request might look something like this:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

See Also

Serving Client Requests (see page 2249)

Responding to Client Requests (see page 2249)

HTTP Request Header Information (see page 2252)

Parts of a Uniform Resource Locator (see page 2253)

3.2.5.1.3.2 HTTP Server Activity

The client/server nature of Web browsers is deceptively simple. To most users, retrieving information on the World Wide Web is a simple procedure: click on a link, and the information appears on the screen. More knowledgeable users have some understanding of the nature of HTML syntax and the client/server nature of the protocols used. This is usually sufficient for the production of simple, page-oriented Web site content. Authors of more complex Web pages have a wide variety of options to automate the collection and presentation of information using HTML.

Before building a Web server application, it is useful to understand how the client issues a request and how the server responds to client requests:

- Composing client requests (see page 2248)
- Serving client requests (see page 2249)
- Responding to client requests (see page 2249)

See Also

Terminology and Standards (see page 2253)

Creating Internet Applications: Overview (see page 2251)

Debugging Server Applications (see page 2245)

3.2.5.1.3.3 Responding to Client Requests

When a Web server application finishes with a client request, it constructs a page of HTML code or other MIME content, and passes it back (via the server) to the client for display. The way the response is sent may differ based on the type of program.

When a DLL finishes, it passes the HTML page and any response information directly back to the server, which passes them back to the client. Creating a Web server application as a DLL reduces system load and resource use by reducing the number of processes and disk accesses necessary to service an individual request.

See Also

Composing Client Requests (see page 2248)

Serving Client Requests (see page 2249)

Types of Web Server Applications (see page 2249)

3.2.5.1.3.4 Serving Client Requests

The Web server receives a client request and can perform any number of actions, based on its configuration. If the server is configured to recognize the /gallery.dll portion of the request as a program, it passes information about the request to that program. The way information about the request is passed to the program depends on the type of Web server application:

- If the program is a Common Gateway Interface (CGI) program, the server passes the information contained in the request directly to the CGI program. The server waits while the program executes. When the CGI program exits, it passes the content directly back to the server.
- If the program is a dynamic-link library (DLL), the server loads the DLL (if necessary) and passes the information contained in the request to the DLL as a structure. The server waits while the program executes. When the DLL exits, it passes the content directly back to the server.

In all cases, the program acts on the request of and performs actions specified by the programmer: accessing databases, doing simple table lookups or calculations, constructing or selecting HTML documents, and so on.

See Also

Composing Client Requests (see page 2248)

Responding to Client Requests (see page 2249)

Types of Web Server Applications (see page 2249)

3.2.5.1.3.5 Types of Web Server Applications

Whether you use Web Broker or WebSnap, you can create five standard types of Web server applications. In addition, you can create a Web Application Debugger executable, which integrates the Web server into your application so that you can debug your application logic. The Web Application Debugger executable is intended only for debugging. When you deploy your application, you should migrate to one of the other five types.

ISAPI and NSAPI

An ISAPI or NSAPI Web server application is a DLL that is loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by the ISAPI/NSAPI application, which creates appropriate request and response objects. Each request message is automatically handled in a separate execution thread.

CGI stand-alone

A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by the CGI application, which creates appropriate request and response objects. Each request message is handled by a separate instance of the application.

Apache

An Apache Web server application is a DLL that is loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by the Apache Web server application, which creates appropriate request and response objects. Each request message is automatically handled in a separate execution thread. You can build your Web server applications using Apache 1 or 2 as your target type.

When you deploy your Apache Web server application, you will need to specify some application-specific information in the Apache configuration files. For example, in Apache 1 projects the default module name is the project name with _module appended to the end. For example, a project named Project1 would have Project1_module as its module name. Similarly, the default content type is the project name with -content appended, and the default handler type is the project name with-handler appended.

These definitions can be changed in the project (.dpr) file when necessary. For example, when you create your project a default module name is stored in the project file. Here is a common example:

```
exports
apache_module name 'Project1_module';
extern "C"
{
Httpd::module __declspec(dllexport) Project1_module;
}
```

Note: When you rename the project during the save process, that name isn't changed automatically. Whenever you rename your project, you must change the module name in your project file to match your project name. The content and handler definitions should change automatically once the module name is changed.

For information on using module, content, and handler definitions in your Apache configuration files, see the documentation on the Apache Web site httpd.apache.org.

Web App Debugger

The server types mentioned above have their advantages and disadvantages for production environments, but none of them is well-suited for debugging. Deploying your application and configuring the debugger can make Web server application debugging far more tedious than debugging other application types.

Fortunately, Web server application debugging doesn't need to be that complicated. The IDE includes a Web App Debugger which makes debugging simple. The Web App Debugger acts like a Web server on your development machine. If you build your Web server application as a Web App Debugger executable, deployment happens automatically during the build process. To debug your application, start it using **Run ▸ Run**. Next, select **Tools ▸ Web App Debugger**, click the default URL and select your application in the Web browser which appears. Your application will launch in the browser window, and you can use the IDE to set breakpoints and obtain debugging information.

When your application is ready to be tested or deployed in a production environment, you can convert your Web App Debugger project to one of the other target types using the steps given below.

Note: When you create a Web App Debugger project, you will need to provide a CoClass Name for your project. This is simply a name used by the Web App Debugger to refer to your application. Most developers use the application's name as the CoClass Name.

Converting Web server application target types

One powerful feature of Web Broker and WebSnap is that they offer several different target server types. The IDE allows you to easily convert from one target type to another.

Because Web Broker and WebSnap have slightly different design philosophies, you must use a different conversion method for each architecture.

To convert your Web Broker application target type

1. Right-click the Web module and choose Add To Repository.
2. In the Add To Repository dialog box, give your Web module a title, text description, Repository page (probably Data Modules), author name, and icon.
3. Choose OK to save your Web module as a template.
4. From the main menu, choose **File** ▶ **New** and select Web Server Application. In the New Web Server Application dialog box, choose the appropriate target type.
5. Delete the automatically generated Web module.
6. From the main menu, choose **File** ▶ **New** and select the template you saved in step 3. This will be on the page you specified in step 2.

To convert a WebSnap application's target type

1. Open your project in the IDE.
2. Display the **Project Manager** using **View** ▶ **Project Manager**. Expand your project so all of its units are visible.
3. In the Project Manager, click the New button to create a new Web server application project. Double-click the WebSnap Application item in the WebSnap tab. Select the appropriate options for your project, including the server type you want to use, then click OK.
4. Expand the new project in the Project Manager. Select any files appearing there and delete them.
5. One at a time, select each file in your project (except for the form file in a Web App Debugger project) and drag it to the new project. When a dialog appears asking if you want to add that file to your new project, click Yes.

See Also

Creating Web Server Applications (see page 2313)

The Structure of a Web Broker Application (see page 2286)

3.2.5.1.4 Creating Internet Applications: Overview

Web server applications extend the functionality and capability of existing Web servers. A Web server application receives HTTP request messages from the Web server, performs any actions requested in those messages, and formulates responses that it passes back to the Web server. Many operations that you can perform with an ordinary application can be incorporated into a Web server application.

The IDE provides two different architectures for developing Web server applications: Web Broker and WebSnap. Although these two architectures are different, WebSnap and Web Broker have many common elements. The WebSnap architecture acts as a superset of Web Broker. It provides additional components and new features like the Preview tab, which allows the content of a page to be displayed without the developer having to run the application. Applications developed with WebSnap can include Web Broker components, whereas applications developed with Web Broker cannot include WebSnap components.

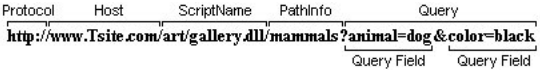
See Also

Working with Sockets (see page 2337)

Creating Active Server Pages (see page 1400)

3.2.5.1.5 Terminology and standards

Topics

Name	Description
HTTP Request Header Information (see page 2252)	HTTP request messages contain many headers that describe information about the client, the target of the request, the way the request should be handled, and any content sent with the request. Each header is identified by a name, such as "Host" followed by a string value. For example, consider the following HTTP request:
Parts of a Uniform Resource Locator (see page 2253)	The Uniform Resource Locator (URL) is a complete description of the location of a resource that is available over the net. It is composed of several parts that may be accessed by an application. These parts are illustrated in the following figure:  The first portion (not technically part of the URL) identifies the protocol (http). This portion can specify other protocols such as https (secure http), ftp, and so on. The Host portion identifies the machine that runs the Web server and Web server application. Although it is not shown in the preceding picture, this portion can override the port... more (see page 2253)
Terminology and Standards (see page 2253)	Many of the protocols that control activity on the Internet are defined in Request for Comment (RFC) documents that are created, updated, and maintained by the Internet Engineering Task Force (IETF), the protocol engineering and development arm of the Internet. There are several important RFCs that you will find useful when writing Internet applications: • RFC822, "Standard for the format of ARPA Internet text messages," describes the structure and content of message headers. • RFC1521, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," describes the method used to encapsulate and transport multipart... more (see page 2253)

3.2.5.1.5.1 HTTP Request Header Information

HTTP request messages contain many headers that describe information about the client, the target of the request, the way the request should be handled, and any content sent with the request. Each header is identified by a name, such as "Host" followed by a string value. For example, consider the following HTTP request:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The first line identifies the request as a GET. A GET request message asks the Web server application to return the content associated with the URI that follows the word GET (in this case /art/gallery.dll/animals?animal=doc&color=black). The last part of the first line indicates that the client is using the HTTP 1.0 standard.

The second line is the Connection header, and indicates that the connection should not be closed once the request is serviced. The third line is the User-Agent header, and provides information about the program generating the request. The next line is the Host header, and provides the Host name and port on the server that is contacted to form the connection. The final line is the Accept header, which lists the media types the client can accept as valid responses.

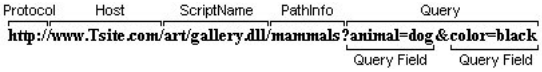
See Also

HTTP Server Activity (see page 2248)

Parts of a Uniform Resource Locator (see page 2253)

3.2.5.1.5.2 Parts of a Uniform Resource Locator

The Uniform Resource Locator (URL) is a complete description of the location of a resource that is available over the net. It is composed of several parts that may be accessed by an application. These parts are illustrated in the following figure:



The first portion (not technically part of the URL) identifies the protocol (http). This portion can specify other protocols such as https (secure http), ftp, and so on.

The Host portion identifies the machine that runs the Web server and Web server application. Although it is not shown in the preceding picture, this portion can override the port that receives messages. Usually, there is no need to specify a port, because the port number is implied by the protocol.

The ScriptName portion specifies the name of the Web server application. This is the application to which the Web server passes messages.

Following the script name is the pathinfo. This identifies the destination of the message within the Web server application. Path info values may refer to directories on the host machine, the names of components that respond to specific messages, or any other mechanism the Web server application uses to divide the processing of incoming messages.

The Query portion contains a set a named values. These values and their names are defined by the Web server application.

URI vs. URL

The URL is a subset of the Uniform Resource Identifier (URI) defined in the HTTP standard, RFC1945. Web server applications frequently produce content from many sources where the final result does not reside in a particular location, but is created as necessary. URIs can describe resources that are not location-specific.

See Also

Server Activity (see page 2248)

Request Header Information (see page 2252)

3.2.5.1.5.3 Terminology and Standards

Many of the protocols that control activity on the Internet are defined in Request for Comment (RFC) documents that are created, updated, and maintained by the Internet Engineering Task Force (IETF), the protocol engineering and development arm of the Internet. There are several important RFCs that you will find useful when writing Internet applications:

- RFC822, "Standard for the format of ARPA Internet text messages," describes the structure and content of message headers.
- RFC1521, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies," describes the method used to encapsulate and transport multipart and multiformat messages.
- RFC1945, "Hypertext Transfer Protocol—HTTP/1.0," describes a transfer mechanism used to distribute collaborative hypermedia documents.

The IETF maintains a library of the RFCs on their Web site, www.ietf.cnri.reston.va.us

These documents include, among other information, details about

- Parts of a Uniform Resource Locator (see page 2253)
- HTTP request header information (see page 2252)
- HTTP server activity (see page 2248)

See Also

Creating Internet Applications: Overview (see page 2251)

Debugging Server Applications (see page 2245)

3.2.5.2 Using IntraWeb (VCL for the Web)

For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

Topics

Name	Description
Getting started with IntraWeb (see page 2254)	This tutorial demonstrates how to use IntraWeb. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.
Using IntraWeb Components (see page 2258)	One of the advantages of IntraWeb is that it uses the same kinds of tools and techniques as regular VCL development. You can build your user interface by dropping components on forms, like you would any other application. There are a number of important differences that you must keep in mind, however. The forms and components used in IntraWeb user interfaces are not the same ones used in non-Web GUI applications. When you create a form or use a component, be sure to use an IntraWeb version instead of a VCL version. Many VCL components have IntraWeb counterparts. Generally, the... more (see page 2258)
Creating Web Server Applications Using IntraWeb (see page 2259)	IntraWeb is a tool which simplifies Web server application development. You can use IntraWeb to build Web server applications exactly the same way you would build traditional GUI applications, using forms. You can write all of your business logic in the Delphi language; IntraWeb will automatically convert program elements to script or HTML when necessary.
Using IntraWeb with Web Broker and WebSnap (see page 2260)	IntraWeb is a powerful tool for developing Web server applications all by itself. Still, there are some things it can't do alone, like create CGI applications. For CGI, you need Web Broker or WebSnap. Also, you may have existing Web Broker and WebSnap applications that you want to extend but not rewrite. You can still take advantage of IntraWeb's design tools by using IntraWeb forms and components in Web Broker or WebSnap projects. You can use IntraWeb to create individual pages instead of entire applications.

3.2.5.2.1 Getting started with IntraWeb

This tutorial demonstrates how to use IntraWeb. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

Topics

Name	Description
Getting Started with IntraWeb (see page 2255)	If you have experience writing GUI applications using Borland's rapid application development tools, then you already have the basic skills you need to start building applications with IntraWeb. The basic design method for the user interface is the same for IntraWeb and regular GUI applications: find the components you need on the **Tool palette** and drop them on a form. Unlike WebSnap's page modules, the appearance of the form mirrors the appearance of the page. The IntraWeb forms and components are distinct from their VCL counterparts, but they are named and arranged similarly. For example, let's say you... more (see page 2255)
Creating a New VCL for the Web Application (see page 2255)	The first step in the process of creating the demo program is to create a new VCL for the Web project. The project will be a stand alone application, but you can convert it to ISAPI/NSAPI or Apache later by changing two lines of code.
Editing the Main Form (see page 2256)	You are now ready to edit the main form to create the Web user interface for your application. For information on creating a new IntraWeb Application, see Creating a New IntraWeb Application (see page 2255).
Writing an Event Handler for the Button (see page 2256)	The form does not yet perform any actions when the user clicks the OK button. For information on editing the main form, see Editing the Main Form (see page 2256).
Running the Completed Application (see page 2257)	You can now test the IntraWeb application. For information on writing the event handler, see Writing an Event Handler for the Button (see page 2256).

Changing VCL for the Web application type (see page 2257)	If you have created a VCL for the Web application and wish to change it to ISAPI/NSAPI or Apache plugins, you can do it with a few simple changes to your code. It is, however, recommended that you start with a standalone application, to ease debugging, and later convert the application to an extension.

3.2.5.2.1.1 Getting Started with IntraWeb

If you have experience writing GUI applications using Borland's rapid application development tools, then you already have the basic skills you need to start building applications with IntraWeb. The basic design method for the user interface is the same for IntraWeb and regular GUI applications: find the components you need on the **Tool palette** and drop them on a form. Unlike WebSnap's page modules, the appearance of the form mirrors the appearance of the page. The IntraWeb forms and components are distinct from their VCL counterparts, but they are named and arranged similarly.

For example, let's say you want to add a button to a form. In an ordinary VCL application, you would find the Button component on the Standard **Tool palette** category and drop it on your form in an appropriate location. In the compiled application, the button appears where you placed it. For an IntraWeb application, the only difference is that you use the IWButton component on the IW Standard category. Even the icons for the two different button components look almost identical. The only difference is an "IW" in the top right corner of the IntraWeb button icon.

Follow the four step tutorial, below, to see how easy it is to build an IntraWeb application. The application you develop in the tutorial asks the user for some input and displays the input in a popup window. The tutorial uses IntraWeb's standalone mode, so the application you create will run without a commercial Web server.

For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

The tutorial includes the following steps:

1. Creating a new IntraWeb application (see page 2255).
2. Editing the main form (see page 2256).
3. Writing an event handler for the button (see page 2256).
4. Running the completed application (see page 2257).

3.2.5.2.1.2 Creating a New VCL for the Web Application

The first step in the process of creating the demo program is to create a new VCL for the Web project. The project will be a stand alone application, but you can convert it to ISAPI/NSAPI or Apache later by changing two lines of code.

To create the new project:

1. Using an external tool (such as Microsoft Windows Explorer), create a directory named Hello in your Projects directory. This is where the project files will be stored. VCL for the Web will set the new project's name to match that of the directory.
2. Choose **File** ▸ **New** ▸ **Other**, then select **VCL for the Web** under either C++Builder Projects or Delphi Projects.
3. Select the **VCL for the Web Application Wizard** and click OK.
4. Select **File** ▸ **Save As** and locate your new Hello directory in the dialog box. Then click OK.

You have just created your VCL for the Web application in the Hello directory. All of its source code files have already been saved. You are now ready to edit the main form to create the Web user interface for your application.

See Also

Changing VCL for the Web application type (see page 2257)

Editing the Main Form (see page 2256)

3.2.5.2.1.3 Editing the Main Form

You are now ready to edit the main form to create the Web user interface for your application.

For information on creating a new IntraWeb Application, see Creating a New IntraWeb Application (see page 2255).

To create the Web user interface for your application:

1. Choose **File** ▸ **Open**, then select IWUnit1.pas and click OK. The main form window (named formMain) should appear in the IDE.
2. Click on the main form window. In the **Object Inspector**, change the form's *Title* property to "What is your name?" This question will appear in the title bar of the Web browser when you run the application and view the page corresponding to the main form.
3. Drop an IWLabel component (found on the IW Standard tab of the **Tool palette**) onto the form. In the **Object Inspector**, change the *Caption* property to "What is your name?" That question should now appear on the form.
4. Drop an IWEdit component onto the form underneath the IWLabel component. Use the **Object Inspector** to make the following changes:

- Empty the contents of the Text property.
- Set the *Name* property to editName.

5. Drop an IWButton component on the form underneath the IWEdit component. Set its *Caption* property to OK. Your form should look similar to this one:

You might want to save all your files before continuing.

For information about writing an event handler for the button, see Writing an Event Handler for the Button (see page 2256).

3.2.5.2.1.4 Writing an Event Handler for the Button

The form does not yet perform any actions when the user clicks the OK button.

For information on editing the main form, see Editing the Main Form (see page 2256).

You will now write an event handler that will display a greeting when the user clicks OK.

1. Double-click the OK button on the form. An empty event handler is created in the editor window, like the one shown here:

```
procedure TformMain.IWButton1Click(Sender: TObject);
begin

end;
```

2. Using the editor, add code to the event handler so it looks like the following:

```
procedure TformMain.IWButton1Click(Sender: TObject);
var s: string;
begin
   s := editName.Text;
   if Length(s) = 0 then
      WebApplication.ShowMessage("Please enter your name.")
   else
   begin
      WebApplication.ShowMessage("Hello, " + s +"!");
      editName.Text := "";
```

```
    end;
end;
```

For information about running the completed application, see Running the Completed Application (see page 2257).

3.2.5.2.1.5 Running the Completed Application

You can now test the IntraWeb application.

For information on writing the event handler, see Writing an Event Handler for the Button (see page 2256).

To test the IntraWeb application:

1. Select **Run** ▸ **Run**. The IntraWeb Application Server (shown below) will appear.

2. In the IntraWeb Application Server, select **Run** ▸ **Execute**. Your Web server application will appear in your default Web browser window. For example, here are the results in a Netscape 6 window:

3. Assume your name is World. Type World in the edit box and click the OK button. A modal dialog box will appear:

When you are finished using your application, you can terminate it by closing the browser window and then closing the IntraWeb Application Server.

3.2.5.2.1.6 Changing VCL for the Web application type

If you have created a VCL for the Web application and wish to change it to ISAPI/NSAPI or Apache plugins, you can do it with a few simple changes to your code.

It is, however, recommended that you start with a standalone application, to ease debugging, and later convert the application to an extension.

To change your standalone application to ISAPI/NSAPI:

1. Open the project source file by selecting **Project** ▸ **View Source**.

2. In the project source file, change the application type from **program** to **library**:
3. In the **uses** clause, replace the **Forms** and **IWMain** units with **ISAPIApp** and **IWInitISAPI**. If you want to use ISAPIThreadPool, include the **ISAPIThreadPool** unit in the **uses** clause.
4. Add the following **exports** section before the program entry point:

```
exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;
```

5. Replace the contents of **begin** ... **end** block with:

```
begin
  IWRun;
end.
```

When you are done the project source file should look like this:

```
library Project1;

uses
  ISAPIApp,
  IWInitISAPI,
  ISAPIThreadPool,
  Unit1 in 'Unit1.pas' {IWForm1: TIWAppForm},
  ServerController in 'ServerController.pas' {IWServerController: TIWServerControllerBase},
  UserSessionUnit in 'UserSessionUnit.pas' {IWUserSession: TIWUserSessionBase};

{$R *.RES}

exports
  GetExtensionVersion,
  HttpExtensionProc,
  TerminateExtension;

begin
  IWRun;
end.
```

To change your standalone application to a service application:

1. Open the project source file by selecting **Project** ▸ **View Source**.
2. In the **uses** clause, replace the **Forms** and **IWMain** units with **IWInitService**.
3. Replace the contents of the **begin** ... **end** block with:

```
begin
  IWRun;
end.
```

See Also

Creating a New VCL for the Web Application (see page 2255)

3.2.5.2.2 Using IntraWeb Components

One of the advantages of IntraWeb is that it uses the same kinds of tools and techniques as regular VCL development. You can build your user interface by dropping components on forms, like you would any other application. There are a number of important differences that you must keep in mind, however. The forms and components used in IntraWeb user interfaces are not the same ones used in non-Web GUI applications. When you create a form or use a component, be sure to use an IntraWeb version instead of a VCL version.

Many VCL components have IntraWeb counterparts. Generally, the IntraWeb components have the same name as their VCL

counterparts, with the letters "IW" prefixed to the name. For example, IWCheckBox is the IntraWeb equivalent of CheckBox. (The name used in source code starts with the letter T, of course, like TIWCheckBox.) On the **Tool palette**, the icons for IntraWeb components are nearly identical to their VCL counterparts, making it easier to find the IntraWeb components you need.

The following table lists VCL components and their IntraWeb counterparts. For more information on these components and how to use them, refer to the IntraWeb help files and other IntraWeb documentation.

VCL and IntraWeb components

VCL component	IntraWeb equivalent	Tool palette category for IntraWeb component
Button	IWButton	IW Standard
CheckBox	IWCheckBox	IW Standard
ComboBox	IWComboBox	IW Standard
DBCheckBox	IWDBCheckBox	IW Data
DBComboBox	IWDBComboBox	IW Data
DBEdit	IWDBEdit	IW Data
DBGrid	IWDBGrid	IW Data
DBImage	IWDBImage	IW Data
DBLabel	IWDBLabel	IW Data
DBListBox	IWDBListBox	IW Data
DBLookupComboBox	IWDBLookupComboBox	IW Data
DBLookupListBox	IWDBLookupListBox	IW Data
DBMemo	IWDBMemo	IW Data
DBNavigator	IWDBNavigator	IW Data
DBText	IWDBText	IW Data
Edit	IWEdit	IW Standard
Image	IWImage or IWImageFile	IW Standard
Label	IWLabel	IW Standard
ListBox	IWListBox	IW Standard
Memo	IWMemo	IW Standard
RadioGroup	IWRadioGroup	IW Standard
Timer	IWTimer	IW Standard
TreeView	IWTreeView	IW Standard

3.2.5.2.3 Creating Web Server Applications Using IntraWeb

IntraWeb is a tool which simplifies Web server application development. You can use IntraWeb to build Web server applications exactly the same way you would build traditional GUI applications, using forms. You can write all of your business logic in the Delphi language; IntraWeb will automatically convert program elements to script or HTML when necessary.

You can use IntraWeb in any of the following modes:

1. **Standalone mode**. IntraWeb uses its own application object type to handle program execution. The application isn't deployed on a commercial server; instead, IntraWeb's own Application Server is used for application deployment.

2. **Application Mode**. The application object is provided by IntraWeb. The application is deployed on a commercial server.
3. **Page mode**. The application object is provided by Web Broker or WebSnap. IntraWeb is used to develop pages. The application is deployed on a commercial server.

IntraWeb applications can be targeted to any of the following server types:

- ISAPI/NSAPI
- Apache versions 1 and 2
- CGI (page mode only)
- Windows services

IntraWeb offers a wide range of browser compatibility. IntraWeb applications automatically detect the user's browser type and generate HTML and script most appropriate for that browser. IntraWeb supports Internet Explorer versions 4 through 6, Netscape 4 and 6, and Mozilla.

For more information, see: http://www.atozed.com/IntraWeb/index.en.aspx.

3.2.5.2.4 Using IntraWeb with Web Broker and WebSnap

IntraWeb is a powerful tool for developing Web server applications all by itself. Still, there are some things it can't do alone, like create CGI applications. For CGI, you need Web Broker or WebSnap. Also, you may have existing Web Broker and WebSnap applications that you want to extend but not rewrite. You can still take advantage of IntraWeb's design tools by using IntraWeb forms and components in Web Broker or WebSnap projects. You can use IntraWeb to create individual pages instead of entire applications.

To create Web pages using IntraWeb tools, use the following steps:

1. Create or open a Web Broker or WebSnap application, and drop a WebDispatcher component on your Web module (Web Broker) or Web application module (WebSnap). The WebDispatcher component is on the Internet tab of the **Tool palette**.
2. Drop an IWModuleController component on your Web module (Web Broker) or Web application module (WebSnap). IWModuleController is on the IW Control category of the **Tool palette**.
3. In WebSnap applications, create a new Web page module if necessary. In the New WebSnap Page dialog, uncheck the New File box in the HTML section before continuing.

   Note: If you create a page module with the New File box checked, you can change the result later. Open the page module's unit file in the editor. Next, change '.html' to an empty string ('') in the WebRequestHandler.AddWebModuleFactory call at the bottom of the unit.

4. Remove any existing page producer components from your Web module (Web Broker) or Web page module (WebSnap).
5. Drop an IWPageProducer component on your Web module or Web page module.
6. Select **File** ▶ **New** ▶ **Other** ▶ **IntraWeb** ▶ **Page Form** to create a new IntraWeb page form.
7. Add an *OnGetForm* event handler by double-clicking the IWPageProducer component on your Web module or Web page module. A new method will appear in the editor window.
8. Connect the IntraWeb form to the Web module or Web page module by adding a line of code to your *OnGetForm* event handler. The code line should be similar to, if not identical to, the following:

```
VForm := TformMain.Create(AWebApplication);
VForm = TformMain->Create(AWebApplication);
```

If necessary, change *TformMain* to the name of your IntraWeb form class. To find the form class name, click on the form. Its name appears next to the form window name in the **Object Inspector**.

9. In the unit file where you changed the event handler, add IWApplication and IWPageForm to the **uses** clause. Also, add the unit containing your form.

3.2.5.3 Using Web Broker

Topics

Name	Description
Generating the content of response messages (see page 2261)	
Using Web Broker (see page 2272)	Web Broker components (located on the Internet tab of the **Tool palette**) enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can programmatically construct HTML or XML documents and transfer them to the client. You can use Web Broker components for cross-platform application development. Frequently, the content of Web pages is drawn from databases. You can use Internet components to automatically manage connections to databases, allowing a single DLL to handle numerous simultaneous, thread-safe database connections. The following sections in this series explain how you use the... more (see page 2272)
The structure of a Web Broker application (see page 2272)	
Creating Web Server Applications with Web Broker (see page 2287)	Web Broker components (located on the Internet tab of the **Tool palette**) enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can programmatically construct HTML or XML documents and transfer them to the client. You can use Web Broker components for cross-platform application development.

3.2.5.3.1 Generating the content of response messages

Topics

Name	Description
Using database information in responses (see page 2262)	
Chaining Page Producers Together (see page 2268)	The replacement text from an OnHTMLTag event handler need not be the final HTML sequence you want to use in the HTTP response message. You may want to use several page producers, where the output from one page producer is the input for the next. The simplest way is to chain the page producers together is to associate each page producer with a separate action item, where all action items have the same *PathInfo* and *MethodType*. The first action item sets the content of the Web response message from its content producer, but its *OnAction* event handler makes sure... more (see page 2268)
Converting HTML-transparent Tags (see page 2269)	The page producer converts the HTML template when you call one of its *Content* methods. When the *Content* method encounters an HTML-transparent tag, it triggers the OnHTMLTag event. You must write an event handler to determine the type of tag encountered, and to replace it with customized content. See Using page producers from an action item (see page 2270) for a simple example of converting HTML-transparent tags. If you do not create an *OnHTMLTag* event handler for the page producer, HTML-transparent tags are replaced with an empty string.
Generating the Content of Response Messages (see page 2269)	Web Broker provides a number of objects to assist your action items in producing content for HTTP response messages. You can use these objects to generate strings of HTML commands that are saved in a file or sent directly back to the Web client. You can write your own content producers, deriving them from TCustomContentProducer or one of its descendants. *TCustomContentProducer* provides a generic interface for creating any MIME type as the content of an HTTP response message. Its descendants include page producers and table producers: • Page producers scan HTML documents for special tags that they replace with customized HTML... more (see page 2269)
HTML Templates (see page 2269)	An HTML template is a sequence of HTML commands and HTML-transparent tags. An HTML-transparent tag has the form

Specifying the HTML Template (see page 2270)	Page producers provide you with many choices in how to specify the HTML template. You can set the HTMLFile property to the name of a file that contains the HTML template. You can set the HTMLDoc property to a TStrings object that contains the HTML template. If you use either the *HTMLFile* property or the *HTMLDoc* property to specify the template, you can generate the converted HTML commands by calling the Content method. In addition, you can call the ContentFromString method to directly convert an HTML template that is a single string which is passed in as a parameter. You... more (see page 2270)
Using Page Producer Components (see page 2270)	Page producers (TPageProducer and its descendants) take an HTML template (see page 2269) and convert it by replacing special HTML-transparent tags with customized HTML code. You can store a set of standard response templates that are filled in by page producers when you need to generate the response to an HTTP request message. You can chain page producers together (see page 2268) to iteratively build up an HTML document by successive refinement of the HTML-transparent tags.
Using Page Producers from an Action Item (see page 2270)	A typical use of a page producer component uses the *HTMLFile* property to specify a file containing an HTML template. The *OnAction* event handler calls the *Content* method to convert the template into a final HTML sequence:
Using Predefined HTML-transparent Tag Names (see page 2271)	While you can create your own HTML-transparent tags to represent any kind of information processed by your page producer, there are several predefined tag names associated with values of the TTag data type. These predefined tag names correspond to HTML commands that are likely to vary over response messages. They are listed in the following table:

3.2.5.3.1.1 Using database information in responses

Topics

Name	Description
Adding a Session to the Web Module (see page 2264)	Console CGI applications are launched in response to HTTP request messages. When working with databases in these types of applications, you can use the default session to manage your database connections, because each request message has its own instance of the application. Each instance of the application has its own distinct, default session. When writing an ISAPI application or an NSAPI application, however, each request message is handled in a separate thread of a single application instance. To prevent the database connections from different threads from interfering with each other, you must give each thread its own session. Each request... more (see page 2264)
Embedding Tables in HTML Documents (see page 2264)	You can embed the HTML table that represents your dataset in a larger document by using the Header and Footer properties of the table producer. Use *Header* to specify everything that comes before the table, and *Footer* to specify everything that comes after the table. You may want to use another content producer (such as a page producer) to create the values for the *Header* and *Footer* properties. If you embed your table in a larger document, you may want to add a caption to the table. Use the Caption and CaptionAlignment properties to give your table a caption.
Representing a Dataset in HTML (see page 2265)	Specialized Content producer components on the Internet palette page supply HTML commands based on the records of a dataset. There are two types of data-aware content producers: • The dataset page producer (see page 2266), which formats the fields of a dataset into the text of an HTML document. • Table producers (see page 2267), which format the records of a dataset as an HTML table.
Specifying the Columns (see page 2265)	If you know the dataset for the table at design time, you can use the Columns editor to customize the columns' field bindings and display attributes. Select the table producer component, and right-click. From the context menu, choose the Columns editor. This lets you add, delete, or rearrange the columns in the table. You can set the field bindings and display properties of individual columns in the **Object Inspector** after selecting them in the Columns editor. If you are getting the name of the dataset from the HTTP request message, you can't bind the fields in the Columns editor at... more (see page 2265)

Specifying the Row Attributes (see page 2265)	Similar to the table attributes, you can specify the alignment and background color of cells in the rows of the table that display data. The RowAttributes property is a THTMLTableRowAttributes object. At design time, specify these properties using the **Object Inspector** by expanding the *RowAttributes* property. You can also specify these properties programmatically at runtime. You can also adjust the number of rows shown in the HTML table by setting the MaxRows property.
Specifying the Table Attributes (see page 2265)	Table producers use the THTMLTableAttributes object to describe the visual appearance of the HTML table that displays the records from the dataset. The *THTMLTableAttributes* object includes properties for the table's width and spacing within the HTML document, and for its background color, border thickness, cell padding, and cell spacing. These properties are all turned into options on the HTML <TABLE> tag created by the table producer. At design time, specify these properties using the **Object Inspector**. Select the table producer object in the **Object Inspector** and expand the TableAttributes property to access the display properties of the *THTMLTableAttributes* object.... more (see page 2265)
Using Dataset Page Producers (see page 2266)	Dataset page producers work like other page producer components: they convert a template that includes HTML-transparent tags into a final HTML representation. They include the special ability, however, of converting tags that have a tag name which matches the name of a field in a dataset into the current value of that field. For more information about using page producers in general, see Using page producer components (see page 2270). To use a dataset page producer, add a TDataSetPageProducer component to your Web module and set its DataSet property to the dataset whose field values should be displayed in the HTML content.... more (see page 2266)
Using Database Information in Responses (see page 2266)	The response to an HTTP request message may include information taken from a database. Specialized content producers on the Internet palette page can generate the HTML to represent the records from a database in an HTML table. To return database information in an HTTP response, you would typically • Add a session to the Web module (see page 2264) • Represent the database information in HTML (see page 2265) As an alternate approach, special components on the InternetExpress category of the **Tool palette** let you build Web servers that are part of a multi-tiered database application. See Building Web applications using InternetExpress (see page 1522) for details.
Using TDataSetTableProducer (see page 2266)	TDataSetTableProducer is a table producer that creates an HTML table for a dataset. Set the DataSet property of *TDataSetTableProducer* to specify the dataset that contains the records you want to display. You do not set the *DataSource* property, as you would for most data-aware objects in a conventional database application. This is because *TDataSetTableProducer* generates its own data source internally. You can set the value of *DataSet* at design time if your Web application always displays records from the same dataset. You must set the *DataSet* property at runtime if you are basing the dataset on the information in the... more (see page 2266)
Using TQueryTableProducer (see page 2267)	You can produce an HTML table to display the results of a query, where the parameters of the query come from the HTTP request message. Specify the TQuery object that uses those parameters as the Query property of a TQueryTableProducer component. If the request message is a GET request, the parameters of the query come from the *Query* fields of the URL that was given as the target of the HTTP request message. If the request message is a POST request, the parameters of the query come from the content of the request message. When you call the Content method... more (see page 2267)

Using Table Producers (see page 2267)	The Internet palette page includes two components that create an HTML table to represent the records of a dataset: • Dataset table producers (see page 2266), which format the fields of a dataset into the text of an HTML document. • Query table producers (see page 2267), which runs a query after setting parameters supplied by the request message and formats the resulting dataset as an HTML table. Using either of the two table producers, you can customize the appearance of a resulting HTML table by specifying properties for the table's color, border, separator thickness, and so on. To set the properties of a table producer... more (see page 2267)

3.2.5.3.1.1.1 Adding a Session to the Web Module

Console CGI applications are launched in response to HTTP request messages. When working with databases in these types of applications, you can use the default session to manage your database connections, because each request message has its own instance of the application. Each instance of the application has its own distinct, default session.

When writing an ISAPI application or an NSAPI application, however, each request message is handled in a separate thread of a single application instance. To prevent the database connections from different threads from interfering with each other, you must give each thread its own session.

Each request message in an ISAPI or NSAPI application spawns a new thread. The Web module (see page 2288) for that thread is generated dynamically at runtime. Add a *TSession* object to the Web module to handle the database connections for the thread that contains the Web module.

Separate instances of the Web module are generated for each thread at runtime. Each of those modules contains the session object. Each of those sessions must have a separate name, so that the threads that handle separate request messages do not interfere with each other's database connections. To cause the session objects in each module to dynamically generate unique names for themselves, set the AutoSessionName property of the session object. Each session object will dynamically generate a unique name for itself and set the SessionName property of all datasets in the module to refer to that unique name. This allows all interaction with the database for each request thread to proceed without interfering with any of the other request messages. For more information on sessions, see Managing database sessions (see page 1664).

See Also

Types of Web Server Applications (see page 2249)

Writing Multi-threaded Applications (see page 2231)

3.2.5.3.1.1.2 Embedding Tables in HTML Documents

You can embed the HTML table that represents your dataset in a larger document by using the Header and Footer properties of the table producer. Use *Header* to specify everything that comes before the table, and *Footer* to specify everything that comes after the table.

You may want to use another content producer (such as a page producer) to create the values for the *Header* and *Footer* properties.

If you embed your table in a larger document, you may want to add a caption to the table. Use the Caption and CaptionAlignment properties to give your table a caption.

See Also

Specifying the Table Attributes (see page 2265)

Specifying the Row Attributes (see page 2265)

Specifying the Columns (see page 2265)

Using Page Producer Components (see page 2270)

3.2.5.3.1.1.3 Representing a Dataset in HTML

Specialized Content producer components on the Internet palette page supply HTML commands based on the records of a dataset. There are two types of data-aware content producers:

- The dataset page producer (see page 2266), which formats the fields of a dataset into the text of an HTML document.
- Table producers (see page 2267), which format the records of a dataset as an HTML table.

See Also

Add a session to the Web module (see page 2264)

3.2.5.3.1.1.4 Specifying the Columns

If you know the dataset for the table at design time, you can use the Columns editor to customize the columns' field bindings and display attributes. Select the table producer component, and right-click. From the context menu, choose the Columns editor. This lets you add, delete, or rearrange the columns in the table. You can set the field bindings and display properties of individual columns in the **Object Inspector** after selecting them in the Columns editor.

If you are getting the name of the dataset from the HTTP request message, you can't bind the fields in the Columns editor at design time. However, you can still customize the columns programmatically at runtime, by setting up the appropriate THTMLTableColumn objects and using the methods of the Columns property to add them to the table. If you do not set up the *Columns property*, the table producer creates a default set of columns that match the fields of the dataset and specify no special display characteristics.

See Also

Specifying the Table Attributes (see page 2265)

Specifying the Row Attributes (see page 2265)

Embedding Tables in HTML Documents (see page 2264)

3.2.5.3.1.1.5 Specifying the Row Attributes

Similar to the table attributes, you can specify the alignment and background color of cells in the rows of the table that display data. The RowAttributes property is a THTMLTableRowAttributes object.

At design time, specify these properties using the **Object Inspector** by expanding the *RowAttributes* property. You can also specify these properties programmatically at runtime.

You can also adjust the number of rows shown in the HTML table by setting the MaxRows property.

See Also

Specifying the Table Attributes (see page 2265)

Specifying the Columns (see page 2265)

Embedding Tables in HTML Documents (see page 2264)

3.2.5.3.1.1.6 Specifying the Table Attributes

Table producers use the THTMLTableAttributes object to describe the visual appearance of the HTML table that displays the records from the dataset. The *THTMLTableAttributes* object includes properties for the table's width and spacing within the

HTML document, and for its background color, border thickness, cell padding, and cell spacing. These properties are all turned into options on the HTML <TABLE> tag created by the table producer.

At design time, specify these properties using the **Object Inspector**. Select the table producer object in the **Object Inspector** and expand the TableAttributes property to access the display properties of the *THTMLTableAttributes* object.

You can also specify these properties programmatically at runtime.

See Also

Specifying the Row Attributes (see page 2265)

Specifying the Columns (see page 2265)

Embedding Tables in HTML Documents (see page 2264)

3.2.5.3.1.1.7 Using Dataset Page Producers

Dataset page producers work like other page producer components: they convert a template that includes HTML-transparent tags into a final HTML representation. They include the special ability, however, of converting tags that have a tag name which matches the name of a field in a dataset into the current value of that field. For more information about using page producers in general, see Using page producer components (see page 2270).

To use a dataset page producer, add a TDataSetPageProducer component to your Web module and set its DataSet property to the dataset whose field values should be displayed in the HTML content. Create an HTML template that describes the output of your dataset page producer. For every field value you want to display, include a tag of the form

```
<#FieldName>
```

in the HTML template, where *FieldName* specifies the name of the field in the dataset whose value should be displayed.

When your application calls the *Content*, *ContentFromString*, or *ContentFromStream* method, the dataset page producer substitutes the current field values for the tags that represent fields.

See Also

Using Table Producers (see page 2267)

3.2.5.3.1.1.8 Using Database Information in Responses

The response to an HTTP request message may include information taken from a database. Specialized content producers on the Internet palette page can generate the HTML to represent the records from a database in an HTML table.

To return database information in an HTTP response, you would typically

- Add a session to the Web module (see page 2264)
- Represent the database information in HTML (see page 2265)

As an alternate approach, special components on the InternetExpress category of the **Tool palette** let you build Web servers that are part of a multi-tiered database application. See Building Web applications using InternetExpress (see page 1522) for details.

See Also

Using Page Producer Components (see page 2270)

3.2.5.3.1.1.9 Using TDataSetTableProducer

TDataSetTableProducer is a table producer that creates an HTML table for a dataset. Set the DataSet property of *TDataSetTableProducer* to specify the dataset that contains the records you want to display. You do not set the *DataSource* property, as you would for most data-aware objects in a conventional database application. This is because *TDataSetTableProducer* generates its own data source internally.

You can set the value of *DataSet* at design time if your Web application always displays records from the same dataset. You must set the *DataSet* property at runtime if you are basing the dataset on the information in the HTTP request message.

See Also

Specifying the Table Attributes (see page 2265)

Specifying the Row Attributes (see page 2265)

Specifying the Columns (see page 2265)

Embedding Tables in HTML Documents (see page 2264)

Using TQueryTableProducer (see page 2267)

3.2.5.3.1.1.10 Using TQueryTableProducer

You can produce an HTML table to display the results of a query, where the parameters of the query come from the HTTP request message. Specify the TQuery object that uses those parameters as the Query property of a TQueryTableProducer component.

If the request message is a GET request, the parameters of the query come from the *Query* fields of the URL that was given as the target of the HTTP request message. If the request message is a POST request, the parameters of the query come from the content of the request message.

When you call the Content method of *TQueryTableProducer*, it runs the query, using the parameters it finds in the request object. It then formats an HTML table to display the records in the resulting dataset.

As with any table producer, you can customize the display properties or column bindings of the HTML table, or embed the table in a larger HTML document.

See Also

Specifying the Table Attributes (see page 2265)

Specifying the Row Attributes (see page 2265)

Specifying the Columns (see page 2265)

Embedding Tables in HTML Documents (see page 2264)

Using TDataSetTableProducer (see page 2266)

3.2.5.3.1.1.11 Using Table Producers

The Internet palette page includes two components that create an HTML table to represent the records of a dataset:

- Dataset table producers (see page 2266), which format the fields of a dataset into the text of an HTML document.
- Query table producers (see page 2267), which runs a query after setting parameters supplied by the request message and formats the resulting dataset as an HTML table.

Using either of the two table producers, you can customize the appearance of a resulting HTML table by specifying properties for the table's color, border, separator thickness, and so on. To set the properties of a table producer at design time, double-click the table producer component to display the Response Editor dialog.

- Specifying the table attributes (see page 2265)
- Specifying the row attributes (see page 2265)
- Specifying the columns (see page 2265)
- Embedding tables in HTML documents (see page 2264)

See Also

Using Dataset Page Producers (see page 2266)

3.2.5.3.1.2 Chaining Page Producers Together

The replacement text from an OnHTMLTag event handler need not be the final HTML sequence you want to use in the HTTP response message. You may want to use several page producers, where the output from one page producer is the input for the next.

The simplest way is to chain the page producers together is to associate each page producer with a separate action item, where all action items have the same *PathInfo* and *MethodType*. The first action item sets the content of the Web response message from its content producer, but its *OnAction* event handler makes sure the message is not considered handled. The next action item uses the ContentFromString method of its associated producer to manipulate the content of the Web response message, and so on. Action items after the first one use an *OnAction* event handler such as the following:

```
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
void __fastcall WebModule1::Action2Action(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  Response->Content = PageProducer2->ContentFromString(Response->Content);
}
```

For example, consider an application that returns calendar pages in response to request messages that specify the month and year of the desired page. Each calendar page contains a picture, followed by the name and year of the month between small images of the previous month and next months, followed by the actual calendar. The resulting image looks something like this:



The general form of the calendar is stored in a template file. It looks like this:

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

The *OnHTMLTag* event handler of the first page producer looks up the month and year from the request message. Using that information and the template file, it does the following:

- Replaces <#MonthlyImage> with <#Image Month=January Year=2000>.
- Replaces <#TitleLine> with <#Calendar Month=December Year=1999 Size=Small> January 2000 <#Calendar Month=February Year=2000 Size=Small>.
- Replaces <#MainBody> with <#Calendar Month=January Year=2000 Size=Large>.

The *OnHTMLTag* event handler of the next page producer uses the content produced by the first page producer, and replaces the <#Image Month=January Year=2000> tag with the appropriate HTML <IMG> tag. Yet another page producer resolves the

#Calendar tags with appropriate HTML tables.

See Also

HTML Templates (see page 2269)

Converting HTML-transparent Tags (see page 2269)

3.2.5.3.1.3 Converting HTML-transparent Tags

The page producer converts the HTML template when you call one of its *Content* methods. When the *Content* method encounters an HTML-transparent tag, it triggers the OnHTMLTag event. You must write an event handler to determine the type of tag encountered, and to replace it with customized content. See Using page producers from an action item (see page 2270) for a simple example of converting HTML-transparent tags.

If you do not create an *OnHTMLTag* event handler for the page producer, HTML-transparent tags are replaced with an empty string.

See Also

Using Predefined HTML-transparent Tag Names (see page 2271)

Specifying the HTML Template (see page 2270)

3.2.5.3.1.4 Generating the Content of Response Messages

Web Broker provides a number of objects to assist your action items in producing content for HTTP response messages. You can use these objects to generate strings of HTML commands that are saved in a file or sent directly back to the Web client. You can write your own content producers, deriving them from TCustomContentProducer or one of its descendants.

TCustomContentProducer provides a generic interface for creating any MIME type as the content of an HTTP response message. Its descendants include page producers and table producers:

- Page producers scan HTML documents for special tags that they replace with customized HTML code. They are described in Using page producer components (see page 2270).
- Table producers create HTML commands based on the information in a dataset. They are described in Using database information in responses (see page 2266).

See Also

Action Items (see page 2278)

Accessing Client Request Information (see page 2274)

The Web Dispatcher (see page 2287)

Creating HTTP Response Messages (see page 2283)

3.2.5.3.1.5 HTML Templates

An HTML template is a sequence of HTML commands and HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

The angle brackets (< and >) define the entire scope of the tag. A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The pound sign identifies the string to the page producer as an HTML-transparent tag. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. The tag name can be any valid identifier and identifies the type of conversion the tag represents.

Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form *ParamName=Value*, where there is no space between the parameter name, the equals

symbol (=) and the value. The parameters are separated by whitespace.

The angle brackets (< and >) make the tag transparent to HTML browsers that do not recognize the #TagName construct.

When working with HTML templates, you will

- Optionally, Use predefined HTML-transparent tag Names (see page 2271)
- Specify the HTML template (see page 2270)
- Convert HTML-transparent tags (see page 2269)

See Also

Chaining Page Producers Together (see page 2268)

3.2.5.3.1.6 Specifying the HTML Template

Page producers provide you with many choices in how to specify the HTML template. You can set the HTMLFile property to the name of a file that contains the HTML template. You can set the HTMLDoc property to a TStrings object that contains the HTML template. If you use either the *HTMLFile* property or the *HTMLDoc* property to specify the template, you can generate the converted HTML commands by calling the Content method.

In addition, you can call the ContentFromString method to directly convert an HTML template that is a single string which is passed in as a parameter. You can also call the ContentFromStream method to read the HTML template from a stream. Thus, for example, you could store all your HTML templates in a memo field in a database, and use the *ContentFromStream* method to obtain the converted HTML commands, reading the template directly from a TBlobStream object.

See Also

Using Predefined HTML-transparent Tag Names (see page 2271)

Converting HTML-transparent Tags (see page 2269)

3.2.5.3.1.7 Using Page Producer Components

Page producers (TPageProducer and its descendants) take an HTML template (see page 2269) and convert it by replacing special HTML-transparent tags with customized HTML code. You can store a set of standard response templates that are filled in by page producers when you need to generate the response to an HTTP request message. You can chain page producers together (see page 2268) to iteratively build up an HTML document by successive refinement of the HTML-transparent tags.

See Also

Using Database Information in Responses (see page 2266)

3.2.5.3.1.8 Using Page Producers from an Action Item

A typical use of a page producer component uses the *HTMLFile* property to specify a file containing an HTML template. The *OnAction* event handler calls the *Content* method to convert the template into a final HTML sequence:

```
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
   PageProducer1.HTMLFile := 'Greeting.html';
   Response.Content := PageProducer1.Content;
end;
void __fastcall WebModule1::MyActionEventHandler(TObject *Sender,
   TWebRequest *Request, TWebResponse *Response, bool &Handled)
{
  PageProducer1->HTMLFile = "Greeting.html";
  Response->Content = PageProducer1->Content();
}
```

Greeting.html is a file that contains this HTML template:

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello <#UserName>!  Welcome to our Web site.
</BODY>
</HTML>
```

The *OnHTMLTag* event handler replaces the custom tag (<#UserName>) in the HTML during execution:

```
procedure WebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
   const TagString: string; TagParams: TStrings; var ReplaceText: string);
begin
   if CompareText(TagString,'UserName') = 0 then
      ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
void __fastcall WebModule1::HTMLTagHandler(TObject *Sender, TTag Tag,
   const AnsiString TagString, TStrings *TagParams, AnsiString &ReplaceText)
{
  if (CompareText(TagString,"UserName") == 0)
    ReplaceText = ((TPageProducer *)Sender)->Dispatcher->Request->Content;
}
```

If the content of the request message was the string *Mr. Ed*, the value of *Response.Content* would be

```
<HTML>
<HEAD><TITLE>Our Brand New Web Site</TITLE></HEAD>
<BODY>
Hello Mr. Ed!  Welcome to our Web site.
</BODY>
</HTML>
```

Note: This example uses an *OnAction* event handler to call the content producer and assign the content of the response message. You do not need to write an *OnAction* event handler if you assign the page producer's *HTMLFile* property at design time. In that case, you can simply assign *PageProducer1* as the value of the action item's *Producer* property to accomplish the same effect as the *OnAction* event handler above.

See Also

HTML Templates (see page 2269)

Responding to Request Messages with Action Items (see page 2280)

3.2.5.3.1.9 Using Predefined HTML-transparent Tag Names

While you can create your own HTML-transparent tags to represent any kind of information processed by your page producer, there are several predefined tag names associated with values of the TTag data type. These predefined tag names correspond to HTML commands that are likely to vary over response messages. They are listed in the following table:

Tag Name	TTag value	What the tag should be converted to
Link	*tgLink*	A hypertext link. The result is an HTML sequence beginning with an <A> tag and ending with an </A> tag.
Image	*tgImage*	A graphic image. The result is an HTML <IMG> tag.
Table	*tgTable*	An HTML table. The result is an HTML sequence beginning with a <TABLE> tag and ending with a </TABLE> tag.
ImageMap	*tgImageMap*	A graphic image with associated hot zones. The result is an HTML sequence beginning with a <MAP> tag and ending with a </MAP> tag.
Object	*tgObject*	An embedded ActiveX object. The result is an HTML sequence beginning with an <OBJECT> tag and ending with an </OBJECT> tag.

Embed	*tgEmbed*	A Netscape-compliant add-in DLL. The result is an HTML sequence beginning with an <EMBED> tag and ending with an </EMBED> tag.

Any other tag name is associated with tgCustom. The page producer supplies no built-in processing of the predefined tag names. They are simply provided to help applications organize the conversion process into many of the more common tasks.

Note: The predefined tag names are case insensitive.

See Also

Specifying the HTML Template (see page 2270)

Converting HTML-transparent Tags (see page 2269)

3.2.5.3.2 Using Web Broker

Web Broker components (located on the Internet tab of the **Tool palette**) enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can programmatically construct HTML or XML documents and transfer them to the client. You can use Web Broker components for cross-platform application development.

Frequently, the content of Web pages is drawn from databases. You can use Internet components to automatically manage connections to databases, allowing a single DLL to handle numerous simultaneous, thread-safe database connections.

The following sections in this series explain how you use the Web Broker components to create a Web server application.

See Also

The Web Application Object (see page 2288)

Using Data Modules and Remote Data Modules (see page 1912)

3.2.5.3.3 The structure of a Web Broker application

Topics

Name	Description
Accessing client request information (see page 2273)	
Adding Actions to the Dispatcher (see page 2277)	Open the action editor from the **Object Inspector** by clicking the ellipsis on the *Actions* property of the dispatcher. Action items (see page 2278) can be added to the dispatcher by clicking the Add button in the action editor. Add actions to the dispatcher to respond to different request methods or target URIs. You can set up your action items in a variety of ways. You can start with action items that preprocess requests, and end with a default action that checks whether the response is complete and either sends the response or returns an error code. Or, you can add a separate... more (see page 2277)
Action items (see page 2277)	
Dispatching Request Messages (see page 2281)	When the dispatcher receives the client request, it generates a BeforeDispatch event. This provides your application with a chance to preprocess the request message before it is seen by any of the action items (see page 2278). Next, the dispatcher iterates over its list of action items, looking for an entry that matches the PathInfo portion of the request message's target URL and that also provides the service specified as the method of the request message. It does this by comparing the PathInfo and MethodType properties of the *TWebRequest* object with the property of the same name on the action item. When... more (see page 2281)
Creating HTTP response messages (see page 2282)	

The Structure of a Web Broker Application (see page 2286)	When the Web application (see page 2288) receives an HTTP request message, it creates a TWebRequest object to represent the HTTP request message, and a TWebResponse object to represent the response that should be returned. The application then passes these objects to the Web dispatcher (either the Web module or a *TWebDispatcher* component). The Web dispatcher (see page 2287) controls the flow of the Web server application. The dispatcher maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The dispatcher identifies the appropriate action items or auto-dispatching components to handle the HTTP request message, and... more (see page 2286)
The Web dispatcher (see page 2287)	

3.2.5.3.3.1 Accessing client request information

Topics

Name	Description
Accessing Client Request Information (see page 2274)	When an HTTP request message is received by the Web server application, the headers of the client request are loaded into the properties of an object descended from *TWebRequest*. For example, in NSAPI and ISAPI applications, the request message is encapsulated by a TISAPIRequest object, and console CGI applications use TCGIRequest objects. The properties of the request object are read-only. You can use them to gather all of the information available in the client request, including • Request header information (see page 2274) • The content of the request message (see page 2276)
Properties That Contain Request Header Information (see page 2274)	Most properties in a request object contain information about the request that comes from the HTTP request header. Not every request supplies a value for every one of these properties. Also, some requests may include header fields that are not surfaced in a property of the request object, especially as the HTTP standard continues to evolve. To obtain the value of a request header field that is not surfaced as one of the properties of the request object, use the GetFieldByName method. The request header properties can be categorized by function: • Properties that identify the target (see page 2276) • Properties that describe the (see page 2275)... more (see page 2274)
Properties That Describe the Content (see page 2275)	Most requests do not include any content, as they are requests for information. However, some requests, such as POST requests, provide content that the Web server application is expected to use. The media type of the content is given in the ContentType property, and its length in the ContentLength property. If the content of the message was encoded (for example, for data compression), this information is in the ContentEncoding property. The name and version number of the application that produced the content is specified by the ContentVersion property. The Title property may also provide information about the content.
Properties That Describe the Expected Response (see page 2275)	The Accept property indicates the media types the Web client will accept as the content of the response message. The IfModifiedSince property specifies whether the client only wants information that has changed recently. The Cookie property includes state information (usually added previously by your application) that can modify the response.
Properties That Describe the Web Client (see page 2275)	The request includes several properties that provide information about where the request originated. These include everything from the e-mail address of the sender (the From property), to the URI where the message originated (the Referer or RemoteHost property). If the request contains any content, and that content does not arise from the same URI as the request, the source of the content is given by the DerivedFrom property. You can also determine the IP address of the client (the RemoteAddr property), and the name and version of the application that sent the request (the UserAgent property).
Properties That Identify the Purpose of the Request (see page 2275)	The Method property is a string describing what the request message is asking the server application to do. The HTTP 1.1 standard defines the following methods: ***Predefined tag names***

Properties That Identify the Target (see page 2276)	The full target of the request message is given by the URL property. Usually, this is a URL that can be broken down into the protocol (HTTP), Host (server system), ScriptName (server application), PathInfo (location on the host), and Query. Each of these pieces is surfaced in its own property. The protocol is always HTTP, and the *Host* and *ScriptName* identify the Web server application. The dispatcher uses the *PathInfo* portion when matching action items to request messages. The *Query* is used by some requests to specify the details of the requested information. Its value is also parsed for you... more (see page 2276)
The Content of HTTP Request Messages (see page 2276)	In addition to the header fields, some request messages include a content portion that the Web server application should process in some way. For example, a POST request might include information that should be added to a database accessed by the Web server application. The unprocessed value of the content is given by the Content property. If the content can be parsed into fields separated by ampersands (&), a parsed version is available in the ContentFields property.

3.2.5.3.3.1.1 Accessing Client Request Information

When an HTTP request message is received by the Web server application, the headers of the client request are loaded into the properties of an object descended from *TWebRequest*. For example, in NSAPI and ISAPI applications, the request message is encapsulated by a TISAPIRequest object, and console CGI applications use TCGIRequest objects.

The properties of the request object are read-only. You can use them to gather all of the information available in the client request, including

- Request header information (see page 2274)
- The content of the request message (see page 2276)

See Also

The Web Dispatcher (see page 2287)

Action Items (see page 2278)

Creating HTTP Response Messages (see page 2283)

Generating the Content of Response Messages (see page 2269)

Types of Web Server Applications (see page 2249)

3.2.5.3.3.1.2 Properties That Contain Request Header Information

Most properties in a request object contain information about the request that comes from the HTTP request header. Not every request supplies a value for every one of these properties. Also, some requests may include header fields that are not surfaced in a property of the request object, especially as the HTTP standard continues to evolve. To obtain the value of a request header field that is not surfaced as one of the properties of the request object, use the GetFieldByName method.

The request header properties can be categorized by function:

- Properties that identify the target (see page 2276)
- Properties that describe the Web client (see page 2275)
- Properties that identify the purpose of the request (see page 2275)
- Properties that describe the expected response (see page 2275)
- Properties that describe the content (see page 2275)

See Also

The Content of HTTP Request Messages (see page 2276)

HTTP Request Header Information (see page 2252)

3.2.5.3.3.1.3 Properties That Describe the Content

Most requests do not include any content, as they are requests for information. However, some requests, such as POST requests, provide content that the Web server application is expected to use. The media type of the content is given in the ContentType property, and its length in the ContentLength property. If the content of the message was encoded (for example, for data compression), this information is in the ContentEncoding property. The name and version number of the application that produced the content is specified by the ContentVersion property. The Title property may also provide information about the content.

See Also

Properties That Identify the Target (see page 2276)

Properties That Describe the Web Client (see page 2275)

Properties That Identify the Purpose of the Request (see page 2275)

Properties That Describe the Expected Response (see page 2275)

The Content of HTTP Request Messages (see page 2276)

3.2.5.3.3.1.4 Properties That Describe the Expected Response

The Accept property indicates the media types the Web client will accept as the content of the response message. The IfModifiedSince property specifies whether the client only wants information that has changed recently. The Cookie property includes state information (usually added previously by your application) that can modify the response.

See Also

Properties That Identify the Target (see page 2276)

Properties That Describe the Web Client (see page 2275)

Properties That Identify the Purpose of the Request (see page 2275)

Properties That Describe the Content (see page 2275)

3.2.5.3.3.1.5 Properties That Describe the Web Client

The request includes several properties that provide information about where the request originated. These include everything from the e-mail address of the sender (the From property), to the URI where the message originated (the Referer or RemoteHost property). If the request contains any content, and that content does not arise from the same URI as the request, the source of the content is given by the DerivedFrom property. You can also determine the IP address of the client (the RemoteAddr property), and the name and version of the application that sent the request (the UserAgent property).

See Also

Properties That Identify the Target (see page 2276)

Properties That Identify the Purpose of the Request (see page 2275)

Properties That Describe the Expected Response (see page 2275)

Properties That Describe the Content (see page 2275)

3.2.5.3.3.1.6 Properties That Identify the Purpose of the Request

The Method property is a string describing what the request message is asking the server application to do. The HTTP 1.1 standard defines the following methods:

Predefined tag names

Value	What the message requests
OPTIONS	Information about available communication options.
GET	Information identified by the URL property.
HEAD	Header information from an equivalent GET message, without the content of the response.
POST	The server application to post the data included in the Content property, as appropriate.
PUT	The server application to replace the resource indicated by the URL property with the data included in the Content property.
DELETE	The server application to delete or hide the resource identified by the URL property.
TRACE	The server application to send a loop-back to confirm receipt of the request.

The *Method* property may indicate any other method that the Web client requests of the server.

The Web server application does not need to provide a response for every possible value of *Method*. The HTTP standard does require that it service both GET and HEAD requests, however.

The MethodType property indicates whether the value of *Method* is GET (mtGet), HEAD (mtHead), POST (mtPost), PUT (mtPut) or some other string (mtAny). The dispatcher matches the value of the *MethodType* property with the *MethodType* of each action item.

See Also

Properties That Identify the Target (see page 2276)

Properties That Describe the Web Client (see page 2275)

Properties That Describe the Expected Response (see page 2275)

Properties That Describe the Content (see page 2275)

3.2.5.3.3.1.7 Properties That Identify the Target

The full target of the request message is given by the URL property. Usually, this is a URL that can be broken down into the protocol (HTTP), Host (server system), ScriptName (server application), PathInfo (location on the host), and Query.

Each of these pieces is surfaced in its own property. The protocol is always HTTP, and the *Host* and *ScriptName* identify the Web server application. The dispatcher uses the *PathInfo* portion when matching action items to request messages. The *Query* is used by some requests to specify the details of the requested information. Its value is also parsed for you as the QueryFields property.

See Also

Properties That Describe the Web Client (see page 2275)

Properties That Identify the Purpose of the Request (see page 2275)

Properties That Describe the Expected Response (see page 2275)

Properties That Describe the Content (see page 2275)

Parts of a Uniform Resource Locator (see page 2253)

3.2.5.3.3.1.8 The Content of HTTP Request Messages

In addition to the header fields, some request messages include a content portion that the Web server application should process in some way. For example, a POST request might include information that should be added to a database accessed by the Web server application.

The unprocessed value of the content is given by the Content property. If the content can be parsed into fields separated by ampersands (&), a parsed version is available in the ContentFields property.

See Also

Properties That Contain Request Header Information (see page 2274)

Composing Client Requests (see page 2248)

3.2.5.3.3.2 Adding Actions to the Dispatcher

Open the action editor from the **Object Inspector** by clicking the ellipsis on the *Actions* property of the dispatcher. Action items (see page 2278) can be added to the dispatcher by clicking the Add button in the action editor.

Add actions to the dispatcher to respond to different request methods or target URIs. You can set up your action items in a variety of ways. You can start with action items that preprocess requests, and end with a default action that checks whether the response is complete and either sends the response or returns an error code. Or, you can add a separate action item for every type of request, where each action item completely handles the request.

See Also

Dispatching Request Messages (see page 2281)

3.2.5.3.3.3 Action items

Topics

Name	Description
Action Items (see page 2278)	Each action item (TWebActionItem) performs a specific task in response to a given type of request message. Action items can completely respond to a request or perform part of the response and allow other action items to complete the job. Action items can send the HTTP response message for the request, or simply set up part of the response for other action items to complete. If a response is completed by the action items but not sent, the Web server application sends the response message. Set up action items for your Web server application by • Adding actions to the dispatcher (see page 2277)... more (see page 2278)
Choosing a Default Action Item (see page 2278)	Only one of the action items can be the default action item. The default action item is selected by setting its Default property to *True*. When the *Default* property of an action item is set to *True*, the *Default* property for the previous default action item (if any) is set to *False*. When the dispatcher searches its list of action items to choose one to handle a request, it stores the name of the default action item. If the request has not been fully handled when the dispatcher reaches the end of its list of action items,... more (see page 2278)
Determining When Action Items Fire (see page 2279)	Most properties of the action item determine when the dispatcher selects it to handle an HTTP request message. To set the properties of an action item, you must first bring up the action editor: select the *Actions* property of the dispatcher in the **Object Inspector** and click on the ellipsis. When an action is selected in the action editor, its properties can be modified in the **Object Inspector**. The action item includes properties that specify • The target URL (see page 2281) • The request method type (see page 2280) Other properties that influence when the dispatcher fires an action item are described in • Enabling and disabling (see page 2279)... more (see page 2279)

Enabling and Disabling Action Items (see page 2279)	Each action item has an Enabled property that can be used to enable or disable that action item. By setting *Enabled* to *False*, you disable the action item so that it is not considered by the dispatcher when it looks for an action item to handle a request. A BeforeDispatch event handler can control which action items should process a request by changing the *Enabled* property of the action items before the dispatcher begins matching them to the request message. **Warning:** Changing the *Enabled* property of an action during execution may cause unexpected results for subsequent requests. If the... more (see page 2279)
Responding to Request Messages with Action Items (see page 2280)	The real work of the Web server application is performed by action items when they execute. When the Web dispatcher fires an action item, that action item can respond to the current request message in two ways: • If the action item has an associated producer component as the value of its Producer property, that producer automatically assigns the Content of the response message using its Content method. The Internet page of the **Tool palette** includes a number of content producer components that can help construct an HTML page for the content of the response message. • After the producer has assigned... more (see page 2280)
The Request Method Type (see page 2280)	The MethodType property of an action item indicates what type of request messages it can process. The dispatcher compares the *MethodType* property of an action item to the *MethodType* of the request message. *MethodType* can take one of the following values: ***MethodType values***
The Target URL (see page 2281)	The dispatcher compares the PathInfo property of an action item to the *PathInfo* of the request message. The value of this property should be the path information portion of the URL for all requests that the action item is prepared to handle. For example, given this URL,

3.2.5.3.3.3.1 Action Items

Each action item (TWebActionItem) performs a specific task in response to a given type of request message.

Action items can completely respond to a request or perform part of the response and allow other action items to complete the job. Action items can send the HTTP response message for the request, or simply set up part of the response for other action items to complete. If a response is completed by the action items but not sent, the Web server application sends the response message.

Set up action items for your Web server application by

- Adding actions to the dispatcher (see page 2277)
- Determining when action items fire (see page 2279)
- Responding to request messages with action items (see page 2280)

See Also

The Web Dispatcher (see page 2287)

Accessing Client Request Information (see page 2274)

Creating HTTP Response Messages (see page 2283)

Generating the Content of Response Messages (see page 2269)

3.2.5.3.3.3.2 Choosing a Default Action Item

Only one of the action items can be the default action item. The default action item is selected by setting its Default property to *True*. When the *Default* property of an action item is set to *True*, the *Default* property for the previous default action item (if any) is set to *False*.

When the dispatcher searches its list of action items to choose one to handle a request, it stores the name of the default action item. If the request has not been fully handled when the dispatcher reaches the end of its list of action items, it executes the default action item.

The dispatcher does not check the *PathInfo* or *MethodType* of the default action item. The dispatcher does not even check the Enabled property of the default action item. Thus, you can make sure the default action item is only called at the very end by setting its Enabled property to *False*.

The default action item should be prepared to handle any request that is encountered, even if it is only to return an error code indicating an invalid URI or *MethodType*. If the default action item does not handle the request, no response is sent to the Web client.

Warning: Changing the *Default* property of an action during execution may cause unexpected results for the current request. If the *Default* property of an action that has already been triggered is set to *True*, that action will not be reevaluated and the dispatcher will not trigger that action when it reaches the end of the action list.

See Also

The Target URL (see page 2281)

The Request Method Type (see page 2280)

Enabling and Disabling Action Items (see page 2279)

Dispatching Request Messages (see page 2281)

3.2.5.3.3.3.3 Determining When Action Items Fire

Most properties of the action item determine when the dispatcher selects it to handle an HTTP request message. To set the properties of an action item, you must first bring up the action editor: select the *Actions* property of the dispatcher in the **Object Inspector** and click on the ellipsis. When an action is selected in the action editor, its properties can be modified in the **Object Inspector**.

The action item includes properties that specify

- The target URL (see page 2281)
- The request method type (see page 2280)

Other properties that influence when the dispatcher fires an action item are described in

- Enabling and disabling action items (see page 2279)
- Choosing a default action item (see page 2278)

See Also

Adding Actions to the Dispatcher (see page 2277)

Responding to Request Messages with Action Items (see page 2280)

The Web Dispatcher (see page 2287)

3.2.5.3.3.3.4 Enabling and Disabling Action Items

Each action item has an Enabled property that can be used to enable or disable that action item. By setting *Enabled* to *False*, you disable the action item so that it is not considered by the dispatcher when it looks for an action item to handle a request.

A BeforeDispatch event handler can control which action items should process a request by changing the *Enabled* property of the action items before the dispatcher begins matching them to the request message.

Warning: Changing the *Enabled* property of an action during execution may cause unexpected results for subsequent requests. If the Web server application is a DLL that caches Web modules, the initial state will not be reinitialized for the next request. Use

the *BeforeDispatch* event to ensure that all action items are correctly initialized to their appropriate starting states.

See Also

The Target URL (see page 2281)

The Request Method Type (see page 2280)

Choosing a Default Action Item (see page 2278)

Dispatching Request Messages (see page 2281)

3.2.5.3.3.3.5 Responding to Request Messages with Action Items

The real work of the Web server application is performed by action items when they execute. When the Web dispatcher fires an action item, that action item can respond to the current request message in two ways:

- If the action item has an associated producer component as the value of its Producer property, that producer automatically assigns the Content of the response message using its Content method. The Internet page of the **Tool palette** includes a number of content producer components that can help construct an HTML page for the content of the response message.
- After the producer has assigned any response content (if there is an associated producer), the action item receives an OnAction event. The OnAction event handler is passed the TWebRequest object that represents the HTTP request message and a TWebResponse object to fill with any response information.

If the action item's content can be generated by a single content producer, it is simplest to assign the content producer as the action item's *Producer* property. However, you can always access any content producer from the *OnAction* event handler as well. The *OnAction* event handler allows more flexibility, so that you can use multiple content producers, assign response message properties, and so on.

Both the content-producer component and the *OnAction* event handler can use any objects or runtime library methods to respond to request messages. They can access databases, perform calculations, construct or select HTML documents, and so on. For more information about generating response content using content-producer components, see Generating the content of response messages (see page 2269).

Sending the response

An *OnAction* event handler can send the response back to the Web client by using the methods of the *TWebResponse* object. However, if no action item sends the response to the client, it will still get sent by the Web server application as long as the last action item to look at the request indicates that the request was handled.

Using multiple action items

You can respond to a request from a single action item, or divide the work up among several action items. If the action item does not completely finish setting up the response message, it must signal this state in the *OnAction* event handler by setting the *Handled* parameter to *False*.

If many action items divide up the work of responding to request messages, each setting *Handled to False* so that others can continue, make sure the default action item leaves the Handled parameter set to *True*. Otherwise, no response will be sent to the Web client.

When dividing the work among several action items, either the *OnAction* event handler of the default action item or the *AfterDispatch* event handler of the dispatcher should check whether all the work was done and set an appropriate error code if it is not.

See Also

Adding Actions to the Dispatcher (see page 2277)

The Web Dispatcher (see page 2287)

3.2.5.3.3.3.6 The Request Method Type

The MethodType property of an action item indicates what type of request messages it can process. The dispatcher compares

the *MethodType* property of an action item to the *MethodType* of the request message. *MethodType* can take one of the following values:

MethodType values

Value	Meaning
mtGet	The request is asking for the information associated with the target URI to be returned in a response message.
mtHead	The request is asking for the header properties of a response, as if servicing an mtGet request, but omitting the content of the response.
mtPost	The request is providing information to be posted to the Web application.
mtPut	The request asks that the resource associated with the target URI be replaced by the content of the request message.
mtAny	Matches any request method type, including mtGet, mtHead, mtPut, and mtPost.

See Also

The Target URL (see page 2281)

Enabling and Disabling Action Items (see page 2279)

Choosing a Default Action Item (see page 2278)

HTTP Request Header Information (see page 2252)

3.2.5.3.3.3.7 The Target URL

The dispatcher compares the PathInfo property of an action item to the *PathInfo* of the request message. The value of this property should be the path information portion of the URL for all requests that the action item is prepared to handle. For example, given this URL,

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

and assuming that the `/gallery.dll` part indicates the Web server application, the path information portion is

```
/mammals
```

Use path information to indicate where your Web application should look for information when servicing requests, or to divide you Web application into logical subservices.

See Also

The Request Method Type (see page 2280)

Enabling and Disabling Action Items (see page 2279)

Choosing a Default Action Item (see page 2278)

Parts of a Uniform Resource Locator (see page 2253)

3.2.5.3.3.4 Dispatching Request Messages

When the dispatcher receives the client request, it generates a BeforeDispatch event. This provides your application with a chance to preprocess the request message before it is seen by any of the action items (see page 2278).

Next, the dispatcher iterates over its list of action items, looking for an entry that matches the PathInfo portion of the request message's target URL and that also provides the service specified as the method of the request message. It does this by comparing the PathInfo and MethodType properties of the *TWebRequest* object with the property of the same name on the action item.

When the dispatcher finds an appropriate action item, it causes that action item to fire. When the action item fires, it does one of

the following:

- Fills in the response content and sends the response or signals that the request is completely handled.
- Adds to the response and then allows other action items to complete the job.
- Defers the request to other action items.

After checking all its action items, if the message is not handled the dispatcher checks any specially registered auto-dispatching components that do not use action items. These components are specific to multi-tiered database applications, which are described in Building Web applications using InternetExpress (see page 1522)

If, after checking all the action items and any specially registered auto-dispatching components, the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

If the dispatcher reaches the end of the action list (including the default action, if any) and no actions have been triggered, nothing is passed back to the server. The server simply drops the connection to the client.

If the request is handled by the action items, the dispatcher generates an AfterDispatch event. This provides a final opportunity for your application to check the response that was generated, and make any last minute changes.

See Also

Adding Actions to the Dispatcher (see page 2277)

3.2.5.3.3.5 Creating HTTP response messages

Topics

Name	Description
Creating HTTP Response Messages (see page 2283)	When the Web server application creates a TWebRequest descended object for an incoming HTTP request message, it also creates a corresponding object descended from *TWebResponse* to represent the response message that will be sent in return. For example, in NSAPI and ISAPI applications, the response message is encapsulated by a TISAPIResponse object, and Console CGI applications use TCGIResponse objects. The action items that generate the response to a Web client request fill in the properties of the response object. In some cases, this may be as simple as returning an error code or redirecting the request to another URI. In... more (see page 2283)
Describing the Content (see page 2284)	Several properties describe the content of the response. ContentType gives the media type of the response, and ContentVersion is the version number for that media type. ContentLength gives the length of the response. If the content is encoded (such as for data compression), indicate this with the ContentEncoding property. If the content came from another URI, this should be indicated in the DerivedFrom property. If the value of the content is time-sensitive, the LastModified property and the Expires property indicate whether the value is still valid. The Title property can provide descriptive information about the content.
Describing the Server Application (see page 2284)	Some of the response header properties describe the capabilities of the Web server application. The Allow property indicates the methods to which the application can respond. The Server property gives the name and version number of the application used to generate the response. The Cookies property can hold state information about the client's use of the server application which is included in subsequent request messages.
Filling in the Response Header (see page 2284)	Most of the properties of the TWebResponse object represent the header information of the HTTP response message that is sent back to the Web client. An action item sets these properties from its *OnAction* event handler. Not every response message needs to specify a value for every one of the header properties. The properties that should be set depend on the nature of the request and the status of the response. Use the response object properties for • Indicating the response status (see page 2285) • Indicating the need for client action (see page 2284) • Describing the server application (see page 2284) • Describing the content (see page 2284)

Indicating the Need for Client Action (see page 2284)	When the status code is in the 300-399 range, the client must perform further action before the Web server application can complete its request. If you need to redirect the client to another URI, or indicate that a new URI was created to handle the request, set the Location property. If the client must provide a password before you can proceed, set the WWWAuthenticate property.
Indicating the Response Status (see page 2285)	Every response message must include a status code that indicates the status of the response. You can specify the status code by setting the StatusCode property. The HTTP standard defines a number of standard status codes with predefined meanings. In addition, you can define your own status codes using any of the unused possible values. Each status code is a three-digit number where the most significant digit indicates the class of the response, as follows: • 1xx: Informational (The request was received but has not been fully processed). • 2xx: Success (The request was received, understood, and accepted). • 3xx: Redirection (Further action... more (see page 2285)
Sending the Response (see page 2285)	If you are sure there is no more work to be done in response to a request message, you can send a response directly from an OnAction event handler. The response object provides two methods for sending a response: SendResponse and SendRedirect. Call *SendResponse* to send the response using the specified content and all the header properties of the *TWebResponse* object. If you only need to redirect the Web client to another URI, the *SendRedirect* method is more efficient. If none of the event handlers send the response, the Web application object sends it after the dispatcher finishes. However, if... more (see page 2285)
Setting the Response Content (see page 2285)	For some requests, the response to the request message is entirely contained in the header properties of the response. In most cases, however, action item assigns some content to the response message. This content may be static information stored in a file, or information that was dynamically produced by the action item or its content producer. You can set the content of the response message by using either the Content property or the ContentStream property. The *Content* property is a string. Delphi strings are not limited to text values, so the value of the *Content* property can be a string... more (see page 2285)

3.2.5.3.3.5.1 Creating HTTP Response Messages

When the Web server application creates a TWebRequest descended object for an incoming HTTP request message, it also creates a corresponding object descended from *TWebResponse* to represent the response message that will be sent in return. For example, in NSAPI and ISAPI applications, the response message is encapsulated by a TISAPIResponse object, and Console CGI applications use TCGIResponse objects.

The action items that generate the response to a Web client request fill in the properties of the response object. In some cases, this may be as simple as returning an error code or redirecting the request to another URI. In other cases, this may involve complicated calculations that require the action item to fetch information from other sources and assemble it into a finished form. Most request messages require some response, even if it is only the acknowledgment that a requested action was carried out.

Responding to HTTP requests involves

- Filling in the response header (see page 2284)
- Setting the response content (see page 2285)
- Sending the response (see page 2285)

See Also

Action Items (see page 2278)

Accessing Client Request Information (see page 2274)

The Web Dispatcher (see page 2287)

Generating the Content of Response Messages (see page 2269)

3.2.5.3.3.5.2 Describing the Content

Several properties describe the content of the response. ContentType gives the media type of the response, and ContentVersion is the version number for that media type. ContentLength gives the length of the response. If the content is encoded (such as for data compression), indicate this with the ContentEncoding property. If the content came from another URI, this should be indicated in the DerivedFrom property. If the value of the content is time-sensitive, the LastModified property and the Expires property indicate whether the value is still valid. The Title property can provide descriptive information about the content.

See Also

Indicating the Response Status (see page 2285)

Indicating the Need for Client Action (see page 2284)

Describing the Server Application (see page 2284)

Setting the Response Content (see page 2285)

Generating the Content of Response Messages (see page 2269)

3.2.5.3.3.5.3 Describing the Server Application

Some of the response header properties describe the capabilities of the Web server application. The Allow property indicates the methods to which the application can respond. The Server property gives the name and version number of the application used to generate the response. The Cookies property can hold state information about the client's use of the server application which is included in subsequent request messages.

See Also

Indicating the Response Status (see page 2285)

Indicating the Need for Client Action (see page 2284)

Describing the Content (see page 2284)

3.2.5.3.3.5.4 Filling in the Response Header

Most of the properties of the TWebResponse object represent the header information of the HTTP response message that is sent back to the Web client. An action item sets these properties from its *OnAction* event handler.

Not every response message needs to specify a value for every one of the header properties. The properties that should be set depend on the nature of the request and the status of the response.

Use the response object properties for

- Indicating the response status (see page 2285)
- Indicating the need for client action (see page 2284)
- Describing the server application (see page 2284)
- Describing the content (see page 2284)

See Also

Setting the Response Content (see page 2285)

Sending the Response (see page 2285)

3.2.5.3.3.5.5 Indicating the Need for Client Action

When the status code is in the 300-399 range, the client must perform further action before the Web server application can complete its request. If you need to redirect the client to another URI, or indicate that a new URI was created to handle the

request, set the Location property. If the client must provide a password before you can proceed, set the WWWAuthenticate property.

See Also

Indicating the Response Status (see page 2285)

Describing the Server Application (see page 2284)

Describing the Content (see page 2284)

3.2.5.3.3.5.6 Indicating the Response Status

Every response message must include a status code that indicates the status of the response. You can specify the status code by setting the StatusCode property. The HTTP standard defines a number of standard status codes with predefined meanings. In addition, you can define your own status codes using any of the unused possible values.

Each status code is a three-digit number where the most significant digit indicates the class of the response, as follows:

- 1xx: Informational (The request was received but has not been fully processed).
- 2xx: Success (The request was received, understood, and accepted).
- 3xx: Redirection (Further action by the client is needed to complete the request).
- 4xx: Client Error (The request cannot be understood or cannot be serviced).
- 5xx: Server Error (The request was valid but the server could not handle it).

Associated with each status code is a string that explains the meaning of the status code. This is given by the ReasonString property. For predefined status codes, you do not need to set the *ReasonString* property. If you define your own status codes, you should also set the *ReasonString* property.

See Also

Indicating the Need for Client Action (see page 2284)

Describing the Server Application (see page 2284)

Describing the Content (see page 2284)

3.2.5.3.3.5.7 Sending the Response

If you are sure there is no more work to be done in response to a request message, you can send a response directly from an OnAction event handler. The response object provides two methods for sending a response: SendResponse and SendRedirect. Call *SendResponse* to send the response using the specified content and all the header properties of the *TWebResponse* object. If you only need to redirect the Web client to another URI, the *SendRedirect* method is more efficient.

If none of the event handlers send the response, the Web application object sends it after the dispatcher finishes. However, if none of the action items indicate that they have handled the response, the application will close the connection to the Web client without sending any response.

See Also

Filling in the Response Header (see page 2284)

Setting the Response Content (see page 2285)

3.2.5.3.3.5.8 Setting the Response Content

For some requests, the response to the request message is entirely contained in the header properties of the response. In most cases, however, action item assigns some content to the response message. This content may be static information stored in a file, or information that was dynamically produced by the action item or its content producer.

You can set the content of the response message by using either the Content property or the ContentStream property.

The *Content* property is a string. Delphi strings are not limited to text values, so the value of the *Content* property can be a string of HTML commands, graphics content such as a bit-stream, or any other MIME content type.

Use the *ContentStream* property if the content for the response message can be read from a stream. For example, if the response message should send the contents of a file, use a TFileStream object for the *ContentStream* property. As with the *Content* property, *ContentStream* can provide a string of HTML commands or other MIME content type. If you use the *ContentStream* property, do not free the stream yourself. The Web response object automatically frees it for you.

Note: If the value of the *ContentStream* property is not **nil**, the *Content* property is ignored.

See Also

Filling in the Response Header (see page 2284)

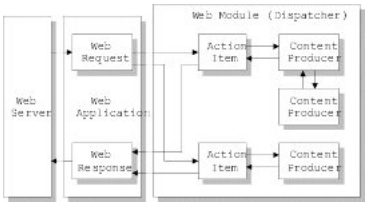
Sending the Response (see page 2285)

Generating the Content of Response Messages (see page 2269)

3.2.5.3.3.6 The Structure of a Web Broker Application

When the Web application (see page 2288) receives an HTTP request message, it creates a TWebRequest object to represent the HTTP request message, and a TWebResponse object to represent the response that should be returned. The application then passes these objects to the Web dispatcher (either the Web module or a *TWebDispatcher* component).

The Web dispatcher (see page 2287) controls the flow of the Web server application. The dispatcher maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The dispatcher identifies the appropriate action items or auto-dispatching components to handle the HTTP request message, and passes the request and response objects to the identified handler so that it can perform any requested actions or formulate a response message.



The action items are responsible for reading the request (see page 2274) and assembling a response message (see page 2283). Specialized content producer components aid the action items in dynamically generating the content of response messages (see page 2269), which can include custom HTML code or other MIME content. The content producers can make use of other content producers or descendants of *THTMLTagAttributes*, to help them create the content of the response message.

If you are creating the Web Client in a multi-tiered database application (see page 1522), your Web server application may include additional, auto-dispatching components that represent database information encoded in XML and database manipulation classes encoded in javascript. If you are creating a server that implements a Web Service (see page 2299), your Web server application may include an auto-dispatching component that passes SOAP-based messages on to an invoker that interprets and executes them. The dispatcher calls on these auto-dispatching components to handle the request message after it has tried all of its action items.

When all action items (or auto-dispatching components) have finished creating the response by filling out the *TWebResponse* object, the dispatcher passes the result back to the Web application. The application sends the response on to the client via the Web server.

See Also

Types of Web Server Applications (see page 2249)

Creating Web Server Applications with Web Broker (see page 2287)

3.2.5.3.3.7 The Web dispatcher

Topics

Name	Description
The Web Dispatcher (see page 2287)	If you are using a Web module (see page 2288), it acts as a Web dispatcher. If you are using a pre-existing data module, you must add a single dispatcher component (TWebDispatcher) to that data module. The dispatcher maintains a collection of action items that know how to handle certain kinds of request messages. When the Web application passes a request object and a response object to the dispatcher, it is responsible for dispatching the request message (see page 2281). Set up the Web dispatcher by adding actions to the dispatcher (see page 2277).

3.2.5.3.3.7.1 The Web Dispatcher

If you are using a Web module (see page 2288), it acts as a Web dispatcher. If you are using a pre-existing data module, you must add a single dispatcher component (TWebDispatcher) to that data module. The dispatcher maintains a collection of action items that know how to handle certain kinds of request messages. When the Web application passes a request object and a response object to the dispatcher, it is responsible for dispatching the request message (see page 2281).

Set up the Web dispatcher by adding actions to the dispatcher (see page 2277).

See Also

Action Items (see page 2278)

Accessing Client Request Information (see page 2274)

Creating HTTP Response Messages (see page 2283)

Generating the Content of Response Messages (see page 2269)

3.2.5.3.4 Creating Web Server Applications with Web Broker

Web Broker components (located on the Internet tab of the **Tool palette**) enable you to create event handlers that are associated with a specific Uniform Resource Identifier (URI). When processing is complete, you can programmatically construct HTML or XML documents and transfer them to the client. You can use Web Broker components for cross-platform application development.

To create a new Web server application using the Web Broker architecture:

1. Select **File** ▶ **New** ▶ **Other**.
2. In the New Items dialog box, select the New tab under Delphi Projects and choose Web Server Application.
3. A dialog box appears, where you can select one of the Web server application types:

- ISAPI and NSAPI: Selecting this type of application sets up your project as a DLL, with the exported methods expected by the Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.
- CGI stand-alone: Selecting this type of application sets up your project as a console application, and adds the required entries to the uses clause of the project file.
- Web Application Debugger stand-alone executable: Selecting this type of application sets up an environment for developing and testing Web server applications. This type of application is not intended for deployment. Choose the type of Web Server Application that communicates with the type of Web Server your application will use. This creates a new project configured to use Internet components.

Topics

Name	Description
The Web Application Object (see page 2288)	The project that is set up for your Web application contains a global variable named *Application*. *Application* is a descendant of *TWebApplication* that is appropriate to the type of application you are creating. It runs in response to HTTP request messages received by the Web server. **Warning:** Do not include the Forms or QForms unit in the project **uses** clause after the CGIApp, ApacheApp, ApacheTwoApp, or ISAPIApp unit. Forms also declares a global variable named *Application*, and if it appears after the CGIApp, ApacheApp, ApacheTwoApp, or ISAPIApp unit, *Application* will... more (see page 2288)
The Web Module (see page 2288)	The Web module (TWebModule) is a descendant of TDataModule and may be used in the same way: to provide centralized control for business rules and non-visual components in the Web application. Add any content producers that your application uses to generate response messages. These can be the built-in content producers such as TPageProducer, TDataSetPageProducer, TDataSetTableProducer, TQueryTableProducer and TInetXPageProducer or descendants of TCustomContentProducer that you have written yourself. If your application generates response messages that include material drawn from databases, you can add data access components or special components for writing a Web server that acts as a client in a (see page 1522)... more (see page 2288)

3.2.5.3.4.1 The Web Application Object

The project that is set up for your Web application contains a global variable named *Application*. *Application* is a descendant of *TWebApplication* that is appropriate to the type of application you are creating. It runs in response to HTTP request messages received by the Web server.

Warning: Do not include the Forms or QForms unit in the project **uses** clause after the CGIApp, ApacheApp, ApacheTwoApp, or ISAPIApp unit. Forms also declares a global variable named *Application*, and if it appears after the CGIApp, ApacheApp, ApacheTwoApp, or ISAPIApp unit, *Application* will be initialized to an object of the wrong type.

See Also

The Web Module (see page 2288)

3.2.5.3.4.2 The Web Module

The Web module (TWebModule) is a descendant of TDataModule and may be used in the same way: to provide centralized control for business rules and non-visual components in the Web application.

Add any content producers that your application uses to generate response messages. These can be the built-in content producers such as TPageProducer, TDataSetPageProducer, TDataSetTableProducer, TQueryTableProducer and TInetXPageProducer or descendants of TCustomContentProducer that you have written yourself. If your application generates response messages that include material drawn from databases, you can add data access components or special components for writing a Web server that acts as a client in a multi-tiered database application (see page 1522).

In addition to storing non-visual components and business rules, the Web module also acts as a Web dispatcher (see page 2287), matching incoming HTTP request messages to action items that generate the responses to those requests.

You may have a data module already that is set up with many of the non-visual components and business rules that you want to use in your Web application. You can replace the Web module with your pre-existing data module. Simply delete the automatically generated Web module and replace it with your data module. Then, add a TWebDispatcher component to your data module, so that it can dispatch request messages to action items, the way a Web module can. If you want to change the way action items are chosen to respond to incoming HTTP request messages, derive a new dispatcher component from TCustomWebDispatcher, and add that to the data module instead.

Your project can contain only one dispatcher. This can either be the Web module that is automatically generated when you create the project, or the TWebDispatcher component that you add to a data module that replaces the Web module. If a second

data module containing a dispatcher is created during execution, the Web server application generates a runtime error.

Note: The Web module that you set up at design time is actually a template. In ISAPI and NSAPI applications, each request message spawns a separate thread, and separate instances of the Web module and its contents are created dynamically for each thread.

Warning: The Web module in a DLL-based Web server application is cached for later reuse to increase response time. The state of the dispatcher and its action list is not reinitialized between requests. Enabling or disabling action items during execution may cause unexpected results when that module is used for subsequent client requests.

See Also

The Web Application Object (see page 2288)

Using Data Modules and Remote Data Modules (see page 1912)

3.2.5.4 Using Web Services

Topics

Name	Description
Using Web Services (see page 2291)	Web Services are self-contained modular applications that can be published and invoked over the Internet. Web Services provide well-defined interfaces that describe the services provided. Unlike Web server applications that generate Web pages for client browsers, Web Services are not designed for direct human interaction. Rather, they are accessed programmatically by client applications. Web Services are designed to allow a loose coupling between client and server. That is, server implementations do not require clients to use a specific platform or programming language. In addition to defining interfaces in a language-neutral fashion, they are designed to allow multiple communications mechanisms as... more (see page 2291)
Understanding Invokable Interfaces (see page 2292)	Servers that support Web Services are built using invokable interfaces. Invokable interfaces are interfaces that are compiled to include runtime type information (RTTI). On the server, this RTTI is used when interpreting incoming method calls from clients so that they can be correctly marshaled. On clients, this RTTI is used to dynamically generate a method table for making calls to the methods of the interface. To create an invokable interface, you need only compile an interface with the {$M+} compiler option. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that... more (see page 2292)
Using Nonscalar Types in Invokable Interfaces (see page 2293)	The Web Services architecture supports the following scalar types for both Delphi and C++:
Registering Nonscalar Types (see page 2294)	Before an invokable interface can use any types other than the built-in scalar types listed in Using nonscalar types in invokable interfaces (see page 2293), the application must register the type with the remotable type registry. To access the remotable type registry, you must add the InvokeRegistry unit to your uses clause. This unit declares a global function, *RemTypeRegistry*, which returns a reference to the remotable type registry. **Note:** On clients, the code to register types with the remotable type registry is generated automatically when you import a WSDL document. For servers, remotable types are registered for you automatically when... more (see page 2294)
Using Remotable Objects (see page 2296)	Use TRemotable as a base class when defining a class to represent a complex data type on an invokable interface. For example, in the case where you would ordinarily pass a record or struct as a parameter, you would instead define a *TRemotable* descendant where every member of the record or struct is a published property on your new class.

Remotable Object Example (see page 2297)	This example shows how to create a remotable object for a parameter on an invokable interface where you would otherwise use an existing class. In this example, the existing class is a string list (*TStringList*). To keep the example small, it does not reproduce the *Objects* property of the string list. Because the new class is not scalar, it descends from *TRemotable* rather than *TRemotableXS*. It includes a published property for every property of the string list you want to communicate between the client and server. Each of these remotable properties corresponds to a remotable type. In... more (see page 2297)
Writing Servers that Support Web Services (see page 2299)	In addition to the invokable interfaces and the classes that implement them, your server requires two components: a dispatcher and an invoker. The dispatcher (THTTPSoapDispatcher) receives incoming SOAP messages and passes them on to the invoker. The invoker (THTTPSOAPPascalInvoker) interprets the SOAP message, identifies the invokable interface it calls, executes the call, and assembles the response message. **Note:** *THTTPSoapDispatcher* and*THTTPSoapPascalInvoker are designed to respond to HTTP messages containing a SOAP request. The underlying architecture is sufficiently general, however, that it can support other protocols with the substitution of different dispatcher and invoker components.* *Once you register your invokable interfaces... more (see page 2299)*
Using the SOAP Application Wizard (see page 2300)	Web Service applications are a special form of Web Server application. Because of this, support for Web Services is built on top of the Web Broker architecture. To understand the code that the SOAP Application wizard generates, therefore, it is helpful to understand the Web Broker architecture. Information about Web Server applications in general, and Web Broker in particular, can be found in Creating Internet server applications (see page 2251) and Using Web Broker (see page 2272). To launch the SOAP application wizard, choose **File** ▶ **New** ▶ **Other**, and on the WebServices page, double-click the Soap Server Application icon. Choose the type of Web server... more (see page 2300)
Adding New Web Services (see page 2301)	To add a new Web Service interface to your server application, choose **File** ▶ **New** ▶ **Other**, and on the WebServices page double-click on the icon labeled SOAP Server Interface. The Add New Web Service wizard lets you specify the name of the invokable interface you want to expose to clients, and generates the code to declare and register the interface and its implementation class. By default, the wizard also generates comments that show sample methods and additional type definitions, to help you get started in editing the generated files.
Using the WSDL Importer (see page 2302)	To use the WSDL importer, choose File|New|Other, and on the WebServices page double-click the icon labeled WSDL importer. In the dialog that appears, specify the file name of a WSDL document (or XML file) or provide the URL where that document is published. **Note:** If you do not know the URL for the WSDL document you want to import, you can browse for one by clicking the button labeled Search UDDI. This launches the UDDI browser. **Tip:** An advantage of using the UDDI browser, even if you know the location of the WSDL document, is that when you locate... more (see page 2302)
Defining and Using SOAP Headers (see page 2303)	The SOAP encoding of a request to your Web Service application and of the response your application sends include a set of header nodes. Some of these, such as the SOAP Action header, are generated and interpreted automatically. However, you can also define your own headers to customize the communication between your server and its clients. Typically, these headers contain information that is associated with the entire invokable interface, or even with the entire application, rather than just the method that is the subject of a single message.
Creating Custom Exception Classes for Web Services (see page 2305)	When your Web Service application raises an exception in the course of trying to execute a SOAP request, it automatically encodes information about that exception in a SOAP fault packet, which it returns instead of the results of the method call. The client application then raises the exception. By default, the client application raises a generic exception of type ERemotableException with the information from the SOAP fault packet. You can transmit additional, application-specific information by deriving an ERemotableException descendant. The values of any published properties you add to the exception class are included in the SOAP fault packet so that... more (see page 2305)

Generating WSDL Documents for a Web Service Application (see page 2306)	To allow client applications to know what Web Services your application makes available, you can publish a WSDL document that describes your invokable interfaces and indicates how to call them. To publish a WSDL document that describes your Web Service, include a TWSDLHTMLPublish component in your Web Module. (The SOAP Server Application wizard adds this component by default.) *TWSDLHTMLPublish* is an auto-dispatching component, which means it automatically responds to incoming messages that request a list of WSDL documents for your Web Service. Use the WebDispatch property to specify the path information of the URL that clients must use to access... more (see page 2306)
Writing Clients for Web Services (see page 2307)	You can write clients that access Web Services that you have written, or any other Web Service that is defined in a WSDL document. There are three steps to writing an application that is the client of a Web Service: • Importing the definitions from a WSDL document (see page 2307). • Obtaining an invokable interface and calling it to invoke the Web Service (see page 2307). • Processing the headers of the SOAP messages that pass between the client and the server (see page 2310).
Importing WSDL Documents (see page 2307)	Before you can use a Web Service, your application must define and register the invokable interfaces and types that are included in the Web Service application. To obtain these definitions, you can import a WSDL document (see page 2302) (or XML file) that defines the service. The WSDL importer creates a unit that defines and registers the interfaces, headers, and types you need to use.
Calling Invokable Interfaces (see page 2307)	To call an invokable interface, your client application must include any definitions of the invokable interfaces and any remotable classes that implement complex types. If the server is written in Delphi or C++Builder, you can use the same units that the server application uses to define and register these interfaces and classes instead of the files generated by importing a WSDL file. Be sure that the unit uses the same namespace URI and SOAPAction header when it registers invokable interfaces. These values can be explicitly specified in the code that registers the interfaces, or it can be automatically generated. If... more (see page 2307)
Processing Headers in Client Applications (see page 2310)	If the Web Service application you are calling expects your client to include any headers in its requests or if its response messages include special headers, your client application needs the definitions of the header classes that correspond to these headers. When you import a WSDL document that describes the Web Service application, the importer automatically generates code to declare these header classes and register them with the remotable type registry. If the server is written in Delphi, you can use the same units that the server application uses to define and register these header classes instead of the files... more (see page 2310)

3.2.5.4.1 Using Web Services

Web Services are self-contained modular applications that can be published and invoked over the Internet. Web Services provide well-defined interfaces that describe the services provided. Unlike Web server applications that generate Web pages for client browsers, Web Services are not designed for direct human interaction. Rather, they are accessed programmatically by client applications.

Web Services are designed to allow a loose coupling between client and server. That is, server implementations do not require clients to use a specific platform or programming language. In addition to defining interfaces in a language-neutral fashion, they are designed to allow multiple communications mechanisms as well.

Support for Web Services is designed to work using SOAP (Simple Object Access Protocol). SOAP is a standard lightweight protocol for exchanging information in a decentralized, distributed environment. It uses XML to encode remote procedure calls and typically uses HTTP as a communications protocol. For more information about SOAP, see the SOAP specification available at

`http://www.w3.org/TR/SOAP/`

Note: Although the components that support Web Services are built to use SOAP and HTTP, the framework is sufficiently

general that it can be expanded to use other encoding and communications protocols.

In addition to letting you build SOAP-based Web Service applications (servers), special components and wizards let you build clients of Web Services that use either a SOAP encoding or a Document Literal style. The Document Literal style is used in .Net Web Services.

Web Service applications publish information on what interfaces are available and how to call them using a WSDL (Web Service Definition Language) document. On the server side, your application can publish a WSDL document that describes your Web Service. On the client side, a wizard or command-line utility can import a published WSDL document, providing you with the interface definitions and connection information you need. If you already have a WSDL document that describes the Web service you want to implement, you can generate the server-side code as well when importing the WSDL document.

The following topics describe support for working with Web Services in greater detail:

- Understanding invokable interfaces (see page 2292)
- Writing servers that support Web Services (see page 2299)
- Writing clients for Web Services (see page 2307)

See Also

Creating Internet Server Applications (see page 2251)

Working with Sockets (see page 2337)

Using Web Services (chapter index) (see page 2289)

3.2.5.4.2 Understanding Invokable Interfaces

Servers that support Web Services are built using invokable interfaces. Invokable interfaces are interfaces that are compiled to include runtime type information (RTTI). On the server, this RTTI is used when interpreting incoming method calls from clients so that they can be correctly marshaled. On clients, this RTTI is used to dynamically generate a method table for making calls to the methods of the interface.

To create an invokable interface, you need only compile an interface with the {$M+} compiler option. The descendant of any invokable interface is also invokable. However, if an invokable interface descends from another interface that is not invokable, your Web Service can only use the methods defined in the invokable interface and its descendants. Methods inherited from the non-invokable ancestors are not compiled with type information and so can't be used as part of the Web Service.

When defining a Web service, you can derive an invokable interface from the base invokable interface, IInvokable. *IInvokable* is defined in the System unit. *IInvokable* is the same as the base interface (*IInterface*), except that it is compiled using the {$M+} compiler option. The {$M+} compiler option ensures that the interface and all its descendants include RTTI.

For example, the following code defines an invokable interface that contains two methods for encoding and decoding numeric values:

```
IEncodeDecode = interface(IInvokable)
['{C527B88F-3F8E-1134-80e0-01A04F57B270}']
    function EncodeValue(Value: Integer): Double; stdcall;
    function DecodeValue(Value: Double): Integer; stdcall;
end;
__interface INTERFACE_UUID("{C527B88F-3F8E-1134-80e0-01A04F57B270}") IEncodeDecode :
    public IInvokable
{
public:
  virtual double __stdcall EncodeValue(int Value) = 0 ;
  virtual int __stdcall DecodeValue(double Value) = 0 ;
};
```

Note: An invokable interface can use overloaded methods, but only if the different overloads can be distinguished by parameter count. That is, one overload must not have the same number of parameters as another, including the possible number of

parameters when default parameters are taken into account.

Before a Web Service application can use this invokable interface, it must be registered with the invocation registry. On the server, the invocation registry entry allows the invoker component (THTTPSOAPPascalInvoker) to identify an implementation class to use for executing interface calls. On client applications, an invocation registry entry allows remote interfaced objects (THTTPRio) to look up information that identifies the invokable interface and supplies information on how to call it.

Typically, your Web Service client or server creates the code to define invokable interfaces either by importing a WSDL document or using the Web Service wizard. By default, when the WSDL importer or Web Service wizard generates an interface, the definition is added to a unit with the same name as the Web Service. This unit includes both the interface definition and code to register the interface with the invocation registry. The invocation registry is a catalog of all registered invokable interfaces, their implementation classes, and any functions that create instances of the implementation classes. It is accessed using the global *InvRegistry* function, which is defined in the InvokeRegistry unit.

The definition of the invokable interface is added to the interface section of the unit, and the code to register the interface goes in the initialization section. The registration code looks like the following:

```
initialization
  InvRegistry.RegisterInterface(TypeInfo(IEncodeDecode));
end.
static void RegTypes()
{
  InvRegistry()->RegisterInterface(__delphirtti(IEncodeDecode), "", "");
}
#pragma startup RegTypes 32
```

Note: The implementation section's uses clause must include the InvokeRegistry unit so that the call to the *InvRegistry* function is defined.

The interfaces of Web Services must have a namespace to identify them among all the interfaces in all possible Web Services. The previous example does not supply a namespace for the interface. When you do not explicitly supply a namespace, the invocation registry automatically generates one for you. This namespace is built from a string that uniquely identifies the application (the *AppNamespacePrefix* variable), the interface name, and the name of the unit in which it is defined. If you do not want to use the automatically-generated namespace, you can specify one explicitly using a second parameter to the *RegisterInterface* call.

You can use the same unit file to define an invokable interface for both client and server applications. If you are doing this, it is a good idea to keep the unit that defines your invokable interfaces separate from the unit in which you write the classes that implement them. Because the generated namespace includes the name of the unit in which the interface is defined, sharing the same unit in both client and server applications enables them to automatically use the same namespace, as long as they both use the same value for the *AppNamespacePrefix* variable.

The preceding example uses only scalar types (integers and doubles) in the methods of the interface. You can use nonscalar types as well, but they require a bit more work.

See Also

Using Nonscalar Types in Invokable Interfaces (see page 2299)

3.2.5.4.3 Using Nonscalar Types in Invokable Interfaces

The Web Services architecture supports the following scalar types for both Delphi and C++:

Boolean	ByteBool	WordBool
LongBool	Char	Byte
ShortInt	SmallInt	Word
Integer	Cardinal	LongInt

Int64	Single	Double
Extended	string	WideString
Currency	TDateTime	Variant

The following additional scalar types are supported for C++:

bool	unsigned short	_int64
char	int	AnsiString
signed char	unsigned	float
unsigned char	long	double
short	unsigned long	long double

You need not do anything special when you use these scalar types on an invokable interface. If your interface includes any properties or methods that use other types, however, your application must register those types with the remotable type registry (see page 2294).

Dynamic arrays can be used in invokable interfaces. They must be registered with the remotable type registry, but this registration happens automatically when you register the interface. The remotable type registry extracts all the information it needs from the type information that the compiler generates.

Note: You should avoid defining multiple dynamic array types with the same element type. Because the compiler treats these as transparent types that can be implicitly cast one to another, it doesn't distinguish their runtime type information. As a result, the remotable type registry can't distinguish the types. This is not a problem for servers, but can result in clients using the wrong type definition. As an alternate approach, you can use remotable clases to represent array types.

Note: The dynamic array types defined in the Types unit are automatically registered for you, so your application does not need to add any special registration code for them. One of these in particular, *TByteDynArray*, deserves special notice because it maps to a 'base64' block of binary data, rather than mapping each array element separately the way the other dynamic array types do.

Enumerated types and types that map directly to one of the automatically-marshaled scalar types can also be used in an invokable interface. As with dynamic array types, they are automatically registered with the remotable type registry.

For any other types, such as static arrays, structs or records, sets, interfaces, or classes, you must map the type to a remotable class (see page 2296). A remotable class is a class that includes runtime type information (RTTI). Your interface must then use the remotable class instead of the corresponding static array, struct or record, set, interface, or class. Any remotable classes you create must be registered with the remotable type registry. As with other types, this registration happens automatically.

See Also

Understanding Invokable Interfaces (see page 2292)

Using Nonscalar Types in Invokable Interfaces (see page 2299)

3.2.5.4.4 Registering Nonscalar Types

Before an invokable interface can use any types other than the built-in scalar types listed in Using nonscalar types in invokable interfaces (see page 2293), the application must register the type with the remotable type registry. To access the remotable type registry, you must add the InvokeRegistry unit to your uses clause. This unit declares a global function, *RemTypeRegistry*, which returns a reference to the remotable type registry.

Note: On clients, the code to register types with the remotable type registry is generated automatically when you import a WSDL document. For servers, remotable types are registered for you automatically when you register an interface that uses

them. You only need to explicitly add code to register types if you want to specify the namespace or type name rather than using the automatically-generated values.

The remotable type registry has two methods that you can use to register types: RegisterXSInfo and RegisterXSClass. The first (*RegisterXSInfo*) lets you register a dynamic array or other type definition. The second (*RegisterXSClass*) is for registering remotable classes that you define to represent other types.

If you are using dynamic arrays or enumerated types, the invocation registry can get the information it needs from the compiler-generated type information. Thus, for example, your interface may use a type such as the following:

```
type
  TDateTimeArray = array of TXSDateTime;
typedef DynamicArray<TXSDateTime> TDateTimeArray;
```

This type is registered automatically when you register the invokable interface. However, if you want to specify the namespace in which the type is defined or the name of the type, you must add code to explicitly register the type using the *RegisterXSInfo* method of the remotable type registry.

The registration goes in the initialization section of the unit where you declare or use the dynamic array:

```
RemTypeRegistry.RegisterXSInfo(TypeInfo(TDateTimeArray), MyNameSpace, 'DTarray', 'DTarray');
void RegTypes()
{
  RemTypeRegistry()->RegisterXSInfo(__arraytypeinfo(TDateTimeArray),
        MyNameSpace, "DTarray", "DTarray");
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

The first parameter of *RegisterXSInfo* is the type information for the type you are registering. The second parameter is the namespace URI for the namespace in which the type is defined. If you omit this parameter or supply an empty string, the registry generates a namespace for you. The third parameter is the name of the type as it appears in native code. If you omit this parameter or supply an empty string, the registry uses the type name from the type information you supplied as the first parameter. The final parameter is the name of the type as it appears in WSDL documents. If you omit this parameter or supply an empty string, the registry uses the native type name (the third parameter).

Registering a remotable class (see page 2296) is similar, except that you supply a class reference rather than a type information pointer. For example, the following line comes from the XSBuiltIns unit. It registers *TXSDateTime*, a *TRemotable* descendant that represents *TDateTime* values:

```
RemClassRegistry.RegisterXSClass(TXSDateTime, XMLSchemaNameSpace, 'dateTime', '',True);
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TXSDateTime), XMLSchemaNameSpace, "dateTime",
"", true);
  InvRegistry()->RegisterInterface(__delphirtti(ITimeServices));
}
```

The first parameter is class reference for the remotable class that represents the type. The second is a uniform resource identifier (URI) that uniquely identifies the namespace of the new class. If you supply an empty string, the registry generates a URI for you. The third and fourth parameters specify the native and external names of the data type your class represents. If you omit the fourth parameter, the type registry uses the third parameter for both values. If you supply an empty string for both parameters, the registry uses the class name. The fifth parameter indicates whether the value of class instances can be transmitted as a string. You can optionally add a sixth parameter (not shown here) to control how multiple references to the same object instance should be represented in SOAP packets.

See Also

Using Remotable Objects (see page 2296)

3.2.5.4.5 Using Remotable Objects

Use TRemotable as a base class when defining a class to represent a complex data type on an invokable interface. For example, in the case where you would ordinarily pass a record or struct as a parameter, you would instead define a *TRemotable* descendant where every member of the record or struct is a published property on your new class.

The following has been deprecated:
You can control whether the published properties of your TRemotable descendant appear as element nodes or attributes in the corresponding SOAP encoding of the type. To make the property an attribute, use the stored directive on the property definition, assigning a value of *AS_ATTRIBUTE*: property MyAttribute: Boolean read FMyAttribute write FMyAttribute stored AS_ATTRIBUTE; __property bool MyAttribute = {read=FMyAttribute, write=FMyAttribute, stored= AS_ATTRIBUTE; If you do not include a stored directive, or if you assign any other value to the stored directive (even a function that returns *AS_ATTRIBUTE*), the property is encoded as a node rather than an attribute.

The *AS_ATTRIBUTE* feature has been deprecated. It still works for legacy code, but the preferred approach is to use the index value of a property. The index property allows you to specify whether a property is an attribute, an unbounded element, an optional element, a text value, or can have a value of NULL. The support for optional elements relies on the stored property attribute, which points to a method that returns *TRUE* if the property has been specified, *FALSE* if not.

The following example illustrates the support for an optional attribute and a text element:

```
class Thumbnail : public TRemotable {
private:
   WideString      FText;
   WideString      FExists;
   bool            FExists_Specified;
   void __fastcall SetExists(int Index, WideString _prop_val)
   {  FExists = _prop_val; FExists_Specified = true;  }
   bool __fastcall Exists_Specified(int Index)
   {  return FExists_Specified;  }
  __published:
   __property WideString       Text = { index=(IS_TEXT), read=FText, write=FText };
   __property WideString     Exists = { index=(IS_ATTR|IS_OPTN), read=FExists,
write=SetExists, stored = Exists_Specified };
};
```

If the value of your new *TRemotable* descendant represents a scalar type in a WSDL document, you should use TRemotableXS as a base class instead. *TRemotableXS* is a *TRemotable* descendant that introduces two methods for converting between your new class and its string representation. Implement these methods by overriding the XSToNative and NativeToXS methods.

For certain commonly-used XML scalar types, the XSBuiltIns unit already defines and registers remotable classes for you. These are listed in the following table:

Remotable classes

XML type	remotable class
dateTime timeInstant	TXSDateTime
date	TXSDate
time	TXSTime
durationtimeDuration	TXSDuration
decimal	TXSDecimal
hexBinary	TXSHexBinary
string	TXSString
boolean	TXSBoolean

integer	TXSInteger
long	TXSLong

Note: You can use *TXSString*, *TXSBoolean*, *TXSInteger* and *TXSLong* for cases where a string, boolean, integer, or long value can be nil (NULL or 0 for C++). You can use the type *TXMLData* when the data sent or received is unspecified. *TXMLData* can be used for passing raw XML—such as a schema—in a SOAP message.

After you define a remotable class, it must be registered with the remotable type registry, as described in Registering nonscalar types (see page 2294). This registration happens automatically on servers when you register the interface that uses the class. On clients, the code to register the class is generated automatically when you import the WSDL document that defines the type. For an example of defining and registering a remotable class, see Remotable object example (see page 2297).

Tip: It is a good idea to implement and register *TRemotable* descendants in a separate unit from the rest of your server application, including from the units that declare and register invokable interfaces. In this way, you can use the type for more than one interface.

Representing attachments

One important *TRemotable* descendant is TSoapAttachment. This class represents an attachment. It can be used as the value of a parameter or the return value of a method on an invokable interface. Attachments are sent with SOAP messages as separate parts in a multipart form.

When a Web Service application or the client of a Web Service receives an attachment, it writes the attachment to a temporary file. *TSoapAttachment* lets you access that temporary file or save its content to a permanent file or stream. When the application needs to send an attachment, it creates an instance of *TSoapAttachment* and assigns its content by specifying the name of a file, supplying a stream from which to read the attachment, or providing a string that represents the content of the attachment.

Managing the lifetime of remotable objects

One issue that arises when using *TRemotable* descendants is the question of when they are created and destroyed. Obviously, the server application must create its own local instance of these objects, because the caller's instance is in a separate process space. To handle this, Web Service applications create a data context for incoming requests. The data context persists while the server handles the request, and is freed after any output parameters are marshaled into a return message. When the server creates local instances of remotable objects, it adds them to the data context, and those instances are then freed along with the data context.

In some cases, you may want to keep an instance of a remotable object from being freed after a method call. For example, if the object contains state information, it may be more efficient to have a single instance that is used for every message call. To prevent the remotable object from being freed along with the data context, change its DataContext property.

See Also

Registering Nonscalar Types (see page 2294)

Remotable Object Example (see page 2297)

3.2.5.4.6 Remotable Object Example

This example shows how to create a remotable object for a parameter on an invokable interface where you would otherwise use an existing class. In this example, the existing class is a string list (*TStringList*). To keep the example small, it does not reproduce the *Objects* property of the string list.

Because the new class is not scalar, it descends from *TRemotable* rather than *TRemotableXS*. It includes a published property for every property of the string list you want to communicate between the client and server. Each of these remotable properties corresponds to a remotable type. In addition, the new remotable class includes methods to convert to and from a string list.

```
TRemotableStringList = class(TRemotable)
```

```
  private
    FCaseSensitive: Boolean;
    FSorted: Boolean;
    FDuplicates: TDuplicates;
    FStrings: TStringDynArray;
  public
    procedure Assign(SourceList: TStringList);
    procedure AssignTo(DestList: TStringList);
  published
    property CaseSensitive: Boolean read FCaseSensitive write FCaseSensitive;
    property Sorted: Boolean read FSorted write FSorted;
    property Duplicates: TDuplicates read FDuplicates write FDuplicates;
    property Strings: TStringDynArray read FStrings write FStrings;
end;
class TRemotableStringList: public TRemotable
{
  private:
    bool FCaseSensitive;
    bool FSorted;
    Classes::TDuplicates FDuplicates;
    System::TStringDynArray FStrings;
  public:
    void __fastcall Assign(Classes::TStringList *SourceList);
    void __fastcall AssignTo(Classes::TStringList *DestList);
__published:
    __property bool CaseSensitive = {read=FCaseSensitive, write=FCaseSensitive};
    __property bool Sorted = {read=FSorted, write=FSorted};
    __property Classes::TDuplicates Duplicates = {read=FDuplicates, write=FDuplicates};
    __property System::TStringDynArray Strings = {read=FStrings, write=FStrings};
}
```

Note that *TRemotableStringList* exists only as a transport class. Thus, although it has a *Sorted* property (to transport the value of a string list's *Sorted* property), it does not need to sort the strings it stores, it only needs to record whether the strings should be sorted. This keeps the implementation very simple. You only need to implement the *Assign* and *AssignTo* methods, which convert to and from a string list:

```
procedure TRemotableStringList.Assign(SourceList: TStrings);
var I: Integer;
begin
  SetLength(Strings, SourceList.Count);
  for I := 0 to SourceList.Count - 1 do
    Strings[I] := SourceList[I];
  CaseSensitive := SourceList.CaseSensitive;
  Sorted := SourceList.Sorted;
  Duplicates := SourceList.Duplicates;
end;
procedure TRemotableStringList.AssignTo(DestList: TStrings);
var I: Integer;
begin
  DestList.Clear;
  DestList.Capacity := Length(Strings);
  DestList.CaseSensitive := CaseSensitive;
  DestList.Sorted := Sorted;
  DestList.Duplicates := Duplicates;
  for I := 0 to Length(Strings) - 1 do
    DestList.Add(Strings[I]);
end;
void __fastcall TRemotableStringList::Assign(Classes::TStringList *SourceList)
{
  SetLength(Strings, SourceList->Count);
  for (int i = 0; i < SourceList->Count; i++)
    Strings[i] = SourceList->Strings[i];
  CaseSensitive = SourceList->CaseSensitive;
  Sorted = SourceList->Sorted;
  Duplicates = SourceList->Duplicates;
}
```

```
void __fastcall TRemotableStringList::AssignTo(Classes::TStringList *DestList)
{
  DestList->Clear();
  DestList->Capacity = Length(Strings);
  DestList->CaseSensitive = CaseSensitive;
  DestList->Sorted = Sorted;
  DestList->Duplicates = Duplicates;
  for (int i = 0; i < Length(Strings); i++)
    DestList->Add(Strings[i]);
}
```

Optionally, you may want to register the new remotable class so that you can specify its class name. If you do not register the class, it is registered automatically when you register the interface that uses it. Similarly, if you register the class but not the *TDuplicates* and *TStringDynArray* types that it uses, they are registered automatically. This code shows how to register the *TRemotableStringList* class and the *TDuplicates* type. *TStringDynArray* is registered automatically because it is one of the built-in dynamic array types declared in the Types unit.

This registration code goes in the initialization section of the unit where you define the remotable class:

```
RemClassRegistry.RegisterXSInfo(TypeInfo(TDuplicates), MyNameSpace, 'duplicateFlag');
RemClassRegistry.RegisterXSClass(TRemotableStringList, MyNameSpace, 'stringList', '',False);
void RegTypes()
{
  RemTypeRegistry()->RegisterXSclass(__classid(TRemotableStringList), MyNameSpace,
"stringList", "", false);
}
#pragma startup initServices 32
```

See Also

Registering Nonscalar Types (see page 2294)

Using Remotable Objects (see page 2296)

3.2.5.4.7 Writing Servers that Support Web Services

In addition to the invokable interfaces and the classes that implement them, your server requires two components: a dispatcher and an invoker. The dispatcher (THTTPSoapDispatcher) receives incoming SOAP messages and passes them on to the invoker. The invoker (THTTPSOAPPascalInvoker) interprets the SOAP message, identifies the invokable interface it calls, executes the call, and assembles the response message.

Note: *THTTPSoapDispatcher* and*THTTPSoapPascalInvoker are designed to respond to HTTP messages containing a SOAP request. The underlying architecture is sufficiently general, however, that it can support other protocols with the substitution of different dispatcher and invoker components.*

Once you register your invokable interfaces and their implementation classes, the dispatcher and invoker automatically handle any messages that identify those interfaces in the SOAP Action header of the HTTP request message.

Web services also include a publisher (TWSDLHTMLPublish). Publishers respond to incoming client requests by creating the WSDL documents that describe how to call the Web Services in the application.

Building a Web Service server

RAD Studio provides a wizard to speed development of a Web Service server application.

Use the following steps to build a server application that implements a Web Service:

1. Choose **File** ▶ **New** ▶ **Other** and on the WebServices page, double-click the Soap Server Application icon to launch the SOAP Server Application wizard (see page 2300). The wizard creates a new Web server application that includes the components you need to respond to SOAP requests.
2. When you exit the SOAP Server Application wizard, it asks you if you want to define an interface for your Web Service. If you are creating a Web Service from scratch, click yes, and you will see the Add New Web Service wizard. The wizard adds code

to declare and register a new invokable interface for your Web Service. Edit the generated code to define and implement your Web Service. If you want to add additional interfaces (or you want to define the interfaces at a later time), choose **File** ▸ **New** ▸ **Other**, and on the WebServices page, double-click the SOAP Web Service interface icon. For details on using the Add New Web Service wizard and completing the code it generates, see Adding new Web Services (see page 2301).

3. If you are implementing a Web Service that has already been defined in a WSDL document, you can use the WSDL importer to generate the interfaces, implementation classes, and registration code that your application needs. You need only fill in the body of the methods the importer generates for the implementation classes. For details on using the WSDL importer, see Using the WSDL importer (see page 2302).
4. If you want to use the headers in the SOAP envelope that encodes messages between your application and clients, you can define classes to represent those headers and write code to process them. This is described in Defining and using SOAP headers (see page 2303).
5. If your application raises an exception when attempting to execute a SOAP request, the exception will be automatically encoded in a SOAP fault packet, which is returned instead of the results of the method call. If you want to convey more information than a simple error message, you can create your own exception classes that are encoded and passed to the client. This is described in Creating custom exception classes for Web Services (see page 2305).
6. The SOAP Server Application wizard adds a publisher component (*TWSDLHTMLPublish)* to new Web Service applications. This enables your application to publish WSDL documents that describe your Web Service to clients. For information on the WSDL publisher, see Generating WSDL documents for a Web Service application (see page 2306).

See Also

Understanding Invokable Interfaces (see page 2292)

3.2.5.4.8 Using the SOAP Application Wizard

Web Service applications are a special form of Web Server application. Because of this, support for Web Services is built on top of the Web Broker architecture. To understand the code that the SOAP Application wizard generates, therefore, it is helpful to understand the Web Broker architecture. Information about Web Server applications in general, and Web Broker in particular, can be found in Creating Internet server applications (see page 2251) and Using Web Broker (see page 2272).

To launch the SOAP application wizard, choose **File** ▸ **New** ▸ **Other**, and on the WebServices page, double-click the Soap Server Application icon. Choose the type of Web server application you want to use for your Web Service. For information about different types of Web Server applications, see Types of Web server applications (see page 2249).

The wizard generates a new Web server application that includes a Web module which contains three components:

- An invoker component (THTTPSOAPPascalInvoker). The invoker converts between SOAP messages and the methods of any registered invokable interfaces in your Web Service application.
- A dispatcher component (THTTPSoapDispatcher). The dispatcher automatically responds to incoming SOAP messages and forwards them to the invoker. You can use its *WebDispatch* property to identify the HTTP request messages to which your application responds. This involves setting the *PathInfo* property to indicate the path portion of any URL directed to your application, and the *MethodType* property to indicate the method header for request messages.
- A WSDL publisher (TWSDLHTMLPublish). The WSDL publisher publishes a WSDL document that describes your interfaces and how to call them. The WSDL document tells clients that how to call on your Web Service application. For details on using the WSDL publisher, see Generating WSDL documents for a Web Service application (see page 2306).

The SOAP dispatcher and WSDL publisher are auto-dispatching components. This means they automatically register themselves with the Web module so that it forwards any incoming requests addressed using the path information they specify in their *WebDispatch* properties. If you right-click on the Web module, you can see that in addition to these auto-dispatching components, it has a single Web action item named *DefaultHandler*.

DefaultHandler is the default action item. That is, if the Web module receives a request for which it can't find a handler (can't match the path information), it forwards that message to the default action item. *DefaultHandler* generates a Web page that describes your Web Service. To change the default action, edit this action item's *OnAction* event handler.

See Also

Generating WSDL Documents for a Web Service Application (see page 2306)

Creating Custom Exception Classes for Web Services (see page 2305)

Adding New Web Services (see page 2301)

Defining and Using SOAP Headers (see page 2303)

3.2.5.4.9 Adding New Web Services

To add a new Web Service interface to your server application, choose **File ▸ New ▸ Other**, and on the WebServices page double-click on the icon labeled SOAP Server Interface.

The Add New Web Service wizard lets you specify the name of the invokable interface you want to expose to clients, and generates the code to declare and register the interface and its implementation class. By default, the wizard also generates comments that show sample methods and additional type definitions, to help you get started in editing the generated files.

Editing the generated code

The interface definitions appear in the interface section of the generated unit. This generated unit has the name you specified using the wizard. You will want to change the interface declaration, replacing the sample methods with the methods you are making available to clients.

The wizard generates an implementation class that descends from TInvokableClass and that supports the invokable interface). If you are defining an invokable interface from scratch, you must edit the declaration of the implementation class to match any edits you made to the generated invokable interface.

When adding methods to the invokable interface and implementation class, remember that the methods must only use remotable types. For information on remotable types and invokable interfaces, see Using nonscalar types in invokable interfaces (see page 2293).

Using a different base class

The Add New WebService wizard generates implementation classes that descend from *TInvokableClass*. This is the easiest way to create a new class to implement a Web Service. You can, however, replace this generated class with an implementation class that has a different base class (for example, you may want to use an existing class as a base class.) There are a number of considerations to take into account when you replace the generated implementation class:

- Your new implementation class must support the invokable interface directly. The invocation registry, with which you register invokable interfaces and their implementation classes, keeps track of what class implements each registered interface and makes it available to the invoker component when the invoker needs to call the interface. It can only detect that a class implements an interface if the interface is directly included in the class declaration. It does not detect support an interface if it is inherited along with a base class.
- Your new implementation class must include support for the *IInterface* methods that are part of any interface. This point may seem obvious, but it is an easy one to overlook.
- You must change the generated code that registers the implementation class to include a factory method to create instances of your implementation class.

This last point takes a bit of explanation. When the implementation class descends from *TInvokableClass* and does not replace the inherited constructor with a new constructor that includes one or more parameters, the invocation registry knows how to create instances of the class when it needs them. When you write an implementation class that does not descend from *TInvokableClass*, or when you change the constructor, you must tell the invocation registry how to obtain instances of your implementation class.

You can tell the invocation registry how to obtain instances of your implementation class by supplying it with a factory procedure. Even if you have an implementation class that descends from *TInvokableClass* and that uses the inherited constructor, you may want to supply a factory procedure anyway. For example, you can use a single global instance of your implementation class rather than requiring the invocation registry to create a new instance every time your application receives a call to the invokable interface.

The factory procedure must be of type *TCreateInstanceProc*. It returns an instance of your implementation class. If the procedure creates a new instance, the implementation object should free itself when the reference count on its interface drops

to zero, as the invocation registry does not explicitly free object instances. The following code illustrates another approach, where the factory procedure returns a single global instance of the implementation class:

```
procedure CreateEncodeDecode(out obj: TObject);
begin
  if FEncodeDecode = nil then
  begin
    FEncodeDecode := TEncodeDecode.Create;
    {save a reference to the interface so that the global instance doesn't free itself }
    FEncodeDecodeInterface := FEncodeDecode as IEncodeDecode;
  end;
  obj := FEncodeDecode; { return global instance }
end;
void __fastcall CreateEncodeDecode(System::TObject* &obj)
{
  if (!FEncodeDecode)
  {
    FEncodeDecode = new TEncodeDecodeImpl();
    // save a reference to the interface so that the global instance doesn't free itself
    TEncodeDecodeImpl->QueryInterface(FEncodeDecodeInterface);
  }
  obj = FEncodeDecode;
}
```

Note: In this example, *FEncodeDecodeInterface* is a variable of type *IEncodeDecode*.

You register the factory procedure with an implementation class by supplying it as a second parameter to the call that registers the class with the invocation registry. First, locate the call the wizard generated to register the implementation class. This appears in initialization section of the unit that defines the class. It looks something like the following:

```
InvRegistry.RegisterInvokableClass(TEncodeDecode);
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl));
```

Add a second parameter to this call that specifies the factory procedure:

```
InvRegistry.RegisterInvokableClass(TEncodeDecode, CreateEncodeDecode);
InvRegistry()->RegisterInvokableClass(__classid(TEncodeDecodeImpl), &CreateEncodeDecode);
```

See Also

Generating WSDL Documents for a Web Service Application (see page 2306)

Creating Custom Exception Classes for Web Services (see page 2305)

Using the WSDL Importer (see page 2302)

Using the SOAP Application Wizard (see page 2300)

Understanding Invokable Interfaces (see page 2292)

Defining and Using SOAP Headers (see page 2303)

3.2.5.4.10 Using the WSDL Importer

To use the WSDL importer, choose File|New|Other, and on the WebServices page double-click the icon labeled WSDL importer. In the dialog that appears, specify the file name of a WSDL document (or XML file) or provide the URL where that document is published.

Note: If you do not know the URL for the WSDL document you want to import, you can browse for one by clicking the button labeled Search UDDI. This launches the UDDI browser.

Tip: An advantage of using the UDDI browser, even if you know the location of the WSDL document, is that when you locate the WSDL document using a UDDI description, client applications get fail-over support.

If the WSDL document is on a server that requires authentication (or must be reached using a proxy server that requires authentication), you need to provide a user name and password before the wizard can retrieve the WSDL document. To supply this information, click the Options button and provide the appropriate connection information.

When you click the Next button, the WSDL importer displays the code it generates for every definition in the WSDL document that is compatible with the Web Services framework. That is, it only uses those port types that have a SOAP binding. You can configure the way the importer generates code by clicking the Options button and choosing the options you want.

You can use the WSDL importer when writing either a server or a client application. When writing a server, click the Options button and in the resulting dialog, check the option that tells the importer to generate server code. When you select this option, the importer generates implementation classes for the invokable interfaces, and you need only fill in the bodies of the methods.

Warning: If you import a WSDL document to create a server that implements a Web Service that is already defined, you must still publish your own WSDL document for that service. There may be minor differences in the imported WSDL document and the generated implementation. For example, if the WSDL document or XML schema file uses identifiers that are also keywords, the importer automatically adjusts their names so that the generated code can compile.

When you click Finish, the importer creates new units that define and register invokable interfaces for the operations defined in the document, and that define and register remotable classes for the types that the document defines.

As an alternate approach, you can use the command line WSDL importer instead. For a server, call the command line importer with the -Os option, as follows:

```
WSDLIMP -Os -P -V MyWSDLDoc.wsdl
WSDLIMP -Os -C -V MyWSDLDoc.wsdl
```

For a client application, call the command line importer without the -Os option:

```
WSDLIMP -P -V MyWSDLDoc.wsdl
WSDLIMP -C -V MyWSDLDoc.wsdl
```

Tip: The command line interpreter includes some options that are not available when you use the WSDL importer in the IDE. For details, see the help for WSDLIMP.

See Also

Generating WSDL Documents for a Web Service Application (see page 2306)

Creating Custom Exception Classes for Web Services (see page 2305)

Using the SOAP Application Wizard (see page 2300)

Understanding Invokable Interfaces (see page 2292)

Defining and Using SOAP Headers (see page 2303)

WSDLIMP.EXE (see page 214)

3.2.5.4.11 Defining and Using SOAP Headers

The SOAP encoding of a request to your Web Service application and of the response your application sends include a set of header nodes. Some of these, such as the SOAP Action header, are generated and interpreted automatically. However, you can also define your own headers to customize the communication between your server and its clients. Typically, these headers contain information that is associated with the entire invokable interface, or even with the entire application, rather than just the method that is the subject of a single message.

Defining header classes

For each header you want to define, create a descendant of TSOAPHeader. *TSOAPHeader* is a descendant of *TRemotable*. That is, SOAP header objects are simply special types of remotable objects (see page 2296). As with any remotable object, you can add published properties to your *TSOAPHeader* descendant to represent the information that your header

communicates. Once you have defined a SOAP header class, it must be registered with the remotable type registry (see page 2294). Note that unlike other remotable classes, which are registered automatically when you register an invokable interface that uses them, you must explicitly write code to register your header types.

TSOAPHeader defines two properties that are used to represent attributes of the SOAP header node. These are *MustUnderstand* and *Actor*. When the *MustUnderstand* attribute is *True*, the recipient of a message that includes the header is required to recognize it. If the recipient can't interpret a header with the *MustUnderstand* attribute, it must abort the interpretation of the entire message. An application can safely ignore any headers it does not recognize if their *MustUnderstand* attribute is not set. The use of *MustUnderstand* is qualified by the *Actor* property. *Actor* is a URI that identifies the application to which the header is directed. Thus, for example, if your Web Service application forwards requests on to another service for further processing, some of the headers in client messages may be targeted at that other service. If such a header includes the *MustUnderstand* attribute, you should not abort the request even if your application can't understand the header. Your application is only concerned with those headers that give its URL as the *Actor*.

Sending and receiving headers

Once you have defined and registered header classes, they are available for your application to use. When your application receives a request, the headers on that message are automatically converted into the corresponding *TSOAPHeader* descendants that you have defined. Your application identifies the appropriate header class by matching the name of the header node against the type name you used when you registered the header class or against a name you supply by registering the header class with the invocation registry. Any headers for which the application can't find a match in the remotable type registry are ignored (or, if their *MustUnderstand* attribute is *True*, the application generates a SOAP fault).

You can access the headers your application receives using the ISOAPHeaders interface. There are two ways to obtain this interface: from an instance of *TInvokableClass* or, if you are implementing your invokable interface without using *TInvokableClass*, by calling the global GetSOAPHeaders function.

Use the *Get* method of *ISOAPHeaders* to access the headers by name. For example:

```
TServiceImpl.GetQuote(Symbol: string): Double;
var
  Headers: ISOAPHeaders;
  H: TAuthHeader;
begin
  Headers := Self as ISOAPHeaders;
  Headers.Get(AuthHeader, TSOAPHeader(H)); { Retrieve the authentication header }
  try
    if H = nil then
      raise ERemotableException.Create("SOAP header for authentication required");
    { code here to check name and password }
  finally
    H.Free;
  end;
  { now that user is authenticated, look up and return quote }
end;
```

If you want to include any headers in the response your application generates to a request message, you can use the same interface. *ISOAPHeaders* defines a *Send* method to add headers to the outgoing response. Simply create an instance of each header class that corresponds to a header you want to send, set its properties, and call *Send*:

```
TServiceImpl.GetQuote(Symbol: string): Double;
var
  Headers: ISOAPHeaders;
  H: TQuoteDelay;
  TXSDuration Delay;
begin
  Headers := Self as ISOAPHeaders;
  { code to lookup the quote and set the return value }
  { this code sets the Delay variable to the time delay on the quote }
  H := TQuoteDelay.Create;
  H.Delay := Delay;
  Headers.OwnsSentHeaders := True;
```

```
  Headers.Send(H);
end;
```

Handling scalar-type headers

Some Web Services define and use headers that are simple types (such as an integer or string) rather than a complex structure that corresponds to a remotable type. However, Delphi's support for SOAP headers requires that you use a *TSOAPHeader* descendant to represent header types. You can define header classes for simple types by treating the *TSOAPHeader* class as a holder class. That is, the *TSOAPHeader* descendant has a single published property, which is the type of the actual header. To signal that the SOAP representation does not need to include a node for the *TSOAPHeader* descendant, call the remotable type registry's RegisterSerializeOptions method (after registering the header type) and give your header type an option of *xoSimpleTypeWrapper*.

Communicating the structure of your headers to other applications

If your application defines headers, you need to allow its clients to access those definitions. If those clients are also written in Delphi, you can share the unit that defines and registers your header classes with the client application. However, you may want to let other clients know about the headers you use as well. To enable your application to export information about its header classes, you must register them with the invocation registry. Registering a header class also associates that class with a header name that is defined within a namespace.

Like the code that registers your invokable interface, the code to register a header class for export is added to the initialization section of the unit in which it is defined. Use the global *InvRegistry* function to obtain a reference to the invocation registry and call its *RegisterHeaderClass* method, indicating the interface with which the header is associated:

```
initialization
  InvRegistry.RegisterInterface(TypeInfo(IMyWebService)); {register the interface}
  InvRegistry.RegisterHeaderClass(TypeInfo(IMyWebService), TMyHeaderClass); {and the header}
end.
static void RegTypes()
{
  // register the invokable interface:
  InvRegistry()->RegisterInterface(__delphirtti(IMyService), "", "");
// register the header to be used with it:
  InvRegistry()->RegisterHeaderClass(__delphirtti(IMyService), __classid(TMyHeader));
}
#pragma startup RegTypes 32
```

You can limit the header to a subset of the methods on the interface by subsequent calls to the *RegisterHeaderMethod* method.

Note: The implementation section's uses clause must include the InvokeRegistry unit so that the call to the *InvRegistry* function is defined.

Once you have registered your header class with the invocation registry, its description is added to WSDL documents when you publish your Web Service (see page 2306).

Note: This registration of your header class with the invocation registry is in addition to the registration of that class with the remotable type registry.

See Also

Using Nonscalar Types in Invokable Interfaces (see page 2293)

Generating WSDL Documents for a Web Service Application (see page 2306)

Creating Custom Exception Classes for Web Services (see page 2305)

Processing Headers in Client Applications (see page 2310)

3.2.5.4.12 Creating Custom Exception Classes for Web Services

When your Web Service application raises an exception in the course of trying to execute a SOAP request, it automatically

encodes information about that exception in a SOAP fault packet, which it returns instead of the results of the method call. The client application then raises the exception.

By default, the client application raises a generic exception of type ERemotableException with the information from the SOAP fault packet. You can transmit additional, application-specific information by deriving an ERemotableException descendant. The values of any published properties you add to the exception class are included in the SOAP fault packet so that the client can raise an equivalent exception.

To use an *ERemotableException* descendant, you must register it with the remotable type registry. Thus, in the unit that defines your *ERemotableException* descendant, you must add the InvokeRegistry unit to the uses clause and add a call to the RegisterXSClass method of the object that the global *RemTypeRegistry* function returns.

If the client also defines and registers your *ERemotableException* descendant, then when it receives the SOAP fault packet, it automatically raises an instance of the appropriate exception class, with all properties set to the values in the SOAP fault packet.

To allow clients to import information about your *ERemotableException* descendant, you must register it with the invocation registry as well as the remotable type registry. Add a call to the RegisterException method of the object that the global *InvRegistry* function returns.

See Also

Using Nonscalar Types in Invokable Interfaces (see page 2293)

Generating WSDL Documents for a Web Service Application (see page 2306)

Defining and Using SOAP Headers (see page 2303)

3.2.5.4.13 Generating WSDL Documents for a Web Service Application

To allow client applications to know what Web Services your application makes available, you can publish a WSDL document that describes your invokable interfaces and indicates how to call them.

To publish a WSDL document that describes your Web Service, include a TWSDLHTMLPublish component in your Web Module. (The SOAP Server Application wizard adds this component by default.) *TWSDLHTMLPublish* is an auto-dispatching component, which means it automatically responds to incoming messages that request a list of WSDL documents for your Web Service. Use the WebDispatch property to specify the path information of the URL that clients must use to access the list of WSDL documents. The Web browser can then request the list of WSDL documents by specifying an URL that is made up of the location of the server application followed by the path in the *WebDispatch* property. This URL looks something like the following:

```
http://www.myco.com/MyService.dll/WSDL
```

Tip: If you want to use a physical WSDL file instead, you can display the WSDL document in your Web browser and then save it to generate a WSDL document file.

Note: In addition to the WSDL document, the *THWSDLHTMLPublish* also generates a WS-Inspection document to describe the service for automated tools. The URL for this document looks something like the following:

```
http://www.myco.com/MyService.dll/inspection.wsil
```

It is not necessary to publish the WSDL document from the same application that implements your Web Service. To create an application that simply publishes the WSDL document, omit the code that implements and registers the implementation objects and only include the code that defines and registers invokable interfaces, remotable classes that represent complex types, and any remotable exceptions.

By default, when you publish a WSDL document, it indicates that the services are available at the same URL as the one where you published the WSDL document (but with a different path). If you are deploying multiple versions of your Web Service

application, or if you are publishing the WSDL document from a different application than the one that implements the Web Service, you will need to change the WSDL document so that it includes updated information on where to locate the Web Service.

To change the URL, use the WSDL administrator. The first step is to enable the administrator. You do this by setting the AdminEnabled property of the *TWSDLHTMLPublish* component to true. Then, when you use your browser to display the list of WSDL documents, it includes a button to administer them as well. Use the WSDL administrator to specify the locations (URLs) where you have deployed your Web Service application.

See Also

Using Nonscalar Types in Invokable Interfaces (see page 2293)

Creating Custom Exception Classes for Web Services (see page 2305)

Defining and Using SOAP Headers (see page 2303)

3.2.5.4.14 Writing Clients for Web Services

You can write clients that access Web Services that you have written, or any other Web Service that is defined in a WSDL document. There are three steps to writing an application that is the client of a Web Service:

- Importing the definitions from a WSDL document (see page 2307).
- Obtaining an invokable interface and calling it to invoke the Web Service (see page 2307).
- Processing the headers of the SOAP messages that pass between the client and the server (see page 2310).

See Also

Creating Custom Exception Classes for Web Services (see page 2305)

3.2.5.4.15 Importing WSDL Documents

Before you can use a Web Service, your application must define and register the invokable interfaces and types that are included in the Web Service application. To obtain these definitions, you can import a WSDL document (see page 2302) (or XML file) that defines the service. The WSDL importer creates a unit that defines and registers the interfaces, headers, and types you need to use.

See Also

Calling Invokable Interfaces (see page 2307)

Processing Headers in Client Applications (see page 2310)

3.2.5.4.16 Calling Invokable Interfaces

To call an invokable interface, your client application must include any definitions of the invokable interfaces and any remotable classes that implement complex types.

If the server is written in Delphi or C++Builder, you can use the same units that the server application uses to define and register these interfaces and classes instead of the files generated by importing a WSDL file. Be sure that the unit uses the same namespace URI and SOAPAction header when it registers invokable interfaces. These values can be explicitly specified in the code that registers the interfaces, or it can be automatically generated. If it is automatically generated, the unit that defines the interfaces must have the same name in both client and server, and both client and server must define the global *AppNameSpacePrefix* variable to have the same value.

Once you have the definition of the invokable interface, there are two ways you can obtain an instance to call:

- If you imported a WSDL document, the importer automatically generates a global function that returns the interface, which you can then call.
- You can use a remote interfaced object.

Obtaining an invokable interface from the generated function

The WSDL importer automatically generates a function from which you can obtain the invokable interfaces you imported. For example, if you imported a WSDL document that defined an invokable interface named *IServerInterface*, the generated unit would include the following global function:

```
function GetIServerInterface(UseWSDL: Boolean; Addr: string; HTTPRIO: THTTPRIO):
IServerInterface;
_di_IServerInterface GetIServerInterface(bool UseWSDL, AnsiString Addr, THTTPRIO* HTTPRIO);
```

The generated function takes three parameters: *UseWSDL*, *Addr*, and *HTTPRio*. *UseWSDL* indicates whether to look up the location of the server from a WSDL document (true), or whether the client application supplies the URL for the server (false).

When *UseWSDL* is false, *Addr* is the URL for the Web Service. When *UseWSDL* is true, *Addr* is the URL of a WSDL document that describes the Web Service you are calling. If you supply an empty string, this defaults to the document you imported. This second approach is best if you expect that the URL for the Web Service may change, or that details such as the namespace or SOAP Action header may change. Using this second approach, this information is looked up dynamically at the time your application makes the method call.

Note: For performance reasons, it is best to use a URL instead of a WSDL parameter. For WSDL, the SOAP runtime must perform an `HTTP GET` for the WSDL parameter before invoking a Web Service operation. The *HTTPRio* parameter specifies the Remotable Interface Object to be used to obtain the interface. If you specify a value of NULL for the *HTTPRio* parameter, the method creates a new *THTTPRio* instance. You can specify the *THTTPRio* instance if you want to use the Object Inspector to customize the object and handle events.

Note: The generated function uses an internal remote interfaced object to implement the invokable interface. If you are using this function and find you need to access that underlying remote interfaced object, you can obtain an *IRIOAccess* interface from the invokable interface, and use that to access the remote interfaced object:

```
var
  Interf: IServerInterface;
  RIOAccess: IRIOAccess;
  X: THTTPRIO;
begin
  Intrf := GetIServerInterface(True,
          'http://MyServices.org/scripts/AppServer.dll/wsdl');
  RIOAccess := Intrf as IRIOAccess;
  X := RIOAccess.RIO as THTTPRIO;
_di_EchoUnboundedSoap service = GetEchoUnboundedSoap();
 _di_IRIOAccess rioAccess = service;
 TSOAPConvertOptions options = rioAccess->GetRIO()->Converter->GetOptions();
```

Using a remote interfaced object

If you do not use the global function to obtain the invokable interface you want to call, you can create an instance of THTTPRio for the desired interface:

```
X := THTTPRio.Create(nil);
X = new THTTPRio(NULL);
```

Note: It is important that you do not explicitly destroy the *THTTPRio* instance. If it is created without an *Owner* (as in the previous line of code), it automatically frees itself when its interface is released. If it is created with an *Owner*, the *Owner* is responsible for freeing the *THTTPRio* instance.

Once you have an instance of *THTTPRio*, provide it with the information it needs to identify the server interface and locate the server. There are two ways to supply this information:

If you do not expect the URL for the Web Service or the namespaces and soap Action headers it requires to change, you can simply specify the URL for the Web Service you want to access. *THTTPRio* uses this URL to look up the definition of the interface, plus any namespace and header information, based on the information in the invocation registry. Specify the URL by setting the *URL* property to the location of the server:

```
X.URL := 'http://www.myco.com/MyService.dll/SOAP/IServerInterface';
```

If you want to look up the URL, namespace, or Soap Action header from the WSDL document dynamically at runtime, you can use the *WSDLLocation*, *Service*, and *Port* properties, and it will extract the necessary information from the WSDL document:

```
X.WSDLLocation := 'Cryptography.wsdl';
X.Service := 'Cryptography';
X.Port := 'SoapEncodeDecode';
```

After specifying how to locate the server and identify the interface, you can obtain an interface pointer for the invokable interface from the *THTTPRio* object. You obtain this interface pointer using the as operator. Simply cast the *THTTPRio* instance to the invokable interface:

```
InterfaceVariable := X as IEncodeDecode;
Code := InterfaceVariable.EncodeValue(5);
_di_IEncodeDecode InterfaceVariable;
if (X->QueryInterface(InterfaceVariable) == S_OK)
{
  Code = InterfaceVariable->EncodeValue(5);
}
```

When you obtain the interface pointer, *THTTPRio* creates a vtable for the associated interface dynamically in memory, enabling you to make interface calls.

THTTPRio relies on the invocation registry to obtain information about the invokable interface. If the client application does not have an invocation registry, or if the invokable interface is not registered, *THTTPRio* can't build its in-memory vtable.

Warning: If you assign the interface you obtain from *THTTPRio* to a global variable, you must change that assignment to nil before shutting down your application. For example, if *InterfaceVariable* in the previous code sample is a global variable, rather than stack variable, you must release the interface before the *THTTPRio* object is freed. Typically, this code goes in the *OnDestroy* event handler of the form or data module:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  InterfaceVariable := nil;
end;
void __fastcall TForm1::FormDestroy(TObject *Sender)
{
  InterfaceVariable = NULL;
}
```

The reason you must reassign a global interface variable to nil is because *THTTPRio* builds its vtable dynamically in memory. That vtable must still be present when the interface is released. If you do not release the interface along with the form or data module, it is released when the global variable is freed on shutdown. The memory for global variables may be freed after the form or data module that contains the *THTTPRio* object, in which case the vtable will not be available when the interface is released.

See Also

Importing WSDL Documents (see page 2307)

Processing Headers in Client Applications (see page 2310)

3.2.5.4.17 Processing Headers in Client Applications

If the Web Service application you are calling expects your client to include any headers in its requests or if its response messages include special headers, your client application needs the definitions of the header classes that correspond to these headers. When you import a WSDL document that describes the Web Service application, the importer automatically generates code to declare these header classes and register them with the remotable type registry. If the server is written in Delphi, you can use the same units that the server application uses to define and register these header classes instead of the files generated by importing a WSDL file. Be sure that the unit uses the same namespace URI and SOAPAction header when it registers invokable interfaces. These values can be explicitly specified in the code that registers the interfaces, or it can be automatically generated. If it is automatically generated, the unit that defines the interfaces must have the same name in both client and server, and both client and server must define the global *AppSpacePrefix* variable to have the same value.

Note: For more information about header classes, see Defining and using SOAP headers (see page 2303).

As with a server, client applications use the ISOAPHeaders interface to access incoming headers and add outgoing headers. The remote interfaced object that you use to call invokable interfaces implements the *ISOAPHeaders* interface. However, you can't obtain an *ISOAPHeaders* interface directly from the remote interfaced object. This is because when you try to obtain an interface directly from a remote interfaced object, it generates an in-memory vtable, assuming that the interface is an invokable interface. Thus, you must obtain the *ISOAPHeaders* interface from the invokable interface rather than from the remote interfaced object:

```
var
  Service: IMyService;
  Hdr: TAuthHeader;
  Val: Double;
begin
  Service := HTTPRIO1 as IService;
  Hdr := TAUthHeader.Create;
  try
    Hdr.Name := "Frank Borland";
    Hdr.Password := "SuperDelphi";
    (Service as ISOAPHeaders).Send(Hdr); { add the header to outgoing message }
    Val := Service.GetQuote("BORL"); { invoke the service }
  finally
    Hdr.Free;
  end;
end;
```

See Also

Defining and Using SOAP Headers (see page 2303)

Importing WSDL Documents (see page 2307)

Calling Invokable Interfaces (see page 2307)

3.2.5.5 Using WebSnap

WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

Topics

Name	Description
Creating Web server applications with WebSnap (see page 2311)	

Creating Web Server Applications Using WebSnap (see page 2315)	WebSnap augments Web Broker with additional components, wizards, and views—making it easier to build Web server applications that deliver complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripting makes development and maintenance easier for teams of developers and Web designers. WebSnap allows HTML design experts on your team to make a more effective contribution to Web server development and maintenance. The final product of the WebSnap development process includes a series of scriptable HTML page templates. These pages can be changed using HTML editors that support embedded script tags, like Microsoft FrontPage, or even a simple... more (see page 2315)
Dispatching requests and responses (see page 2316)	
Fundamental WebSnap components (see page 2321)	
Login support (see page 2327)	
Server-side scripting in WebSnap (see page 2332)	

3.2.5.5.1 Creating Web server applications with WebSnap

Topics

Name	Description
Advanced HTML Design (see page 2311)	Using adapters and adapter page producers, WebSnap makes it easy to create scripted HTML pages in your Web server application. You can create a Web front end for your application data using WebSnap tools that may suit all of your needs. One powerful feature of WebSnap, however, is the ability to incorporate Web design expertise from other sources into your application. This section discusses some strategies for expanding the Web server design and maintenance process to include other tools and non-programmer team members. The end products of WebSnap development are your server application and HTML templates for the pages that... more (see page 2311)
Specifying Application Module Components (see page 2312)	Application components provide the Web application's functionality. For example, including an adapter dispatcher component automatically handles HTML form submissions and the return of dynamically generated images. Including a page dispatcher automatically displays the content of a page when the HTTP request pathinfo contains the name of the page. For information on creating web server applications, see Creating Web Server Applications with WebSnap (see page 2313). Selecting the Components button on the new WebSnap application dialog displays another dialog that allows you to select one or more of the Web application module components. The following table contains a brief explanation of the available... more (see page 2312)
Creating Web Server Applications with WebSnap (see page 2313)	If you look at the source code for WebSnap, you will discover that WebSnap comprises hundreds of objects. In fact, WebSnap is so rich in objects and features that you could spend a long time studying its architecture in detail before understanding it completely. Fortunately, you really don't need to understand the whole WebSnap system before you start developing your server application. Here you will learn more about how WebSnap works by creating a new Web server application.
Selecting a Server Type (see page 2314)	Select one of the following types of Web server application, depending on your application's type of Web server. ***Web server application types***
Selecting Web Application Module Options (see page 2314)	If the selected application module type is a page module, you can associate a name with the page by entering a name in the Page Name field in the New WebSnap Application dialog box. At runtime, the instance of this module can be either kept in cache or removed from memory when the request has been serviced. Select either of the options from the Caching field. You can select more page module options by choosing the Page Options button. For information on adding application module components, see Specifying Application Module Components (see page 2312). The Application Module Page Options dialog is displayed... more (see page 2314)

3.2.5.5.1.1 Advanced HTML Design

Using adapters and adapter page producers, WebSnap makes it easy to create scripted HTML pages in your Web server application. You can create a Web front end for your application data using WebSnap tools that may suit all of your needs. One

powerful feature of WebSnap, however, is the ability to incorporate Web design expertise from other sources into your application. This section discusses some strategies for expanding the Web server design and maintenance process to include other tools and non-programmer team members.

The end products of WebSnap development are your server application and HTML templates for the pages that the server produces. The templates include a mixture of scripting and HTML. Once they have been generated initially, they can be edited at any time using any HTML tool you like. (It would be best to use a tool that supports embedded script tags, like Microsoft FrontPage, to ensure that the editor doesn't accidentally damage the script.) The ability to edit template pages outside of the IDE can be used many ways.

After the product has been deployed, you may wish to change the look of the final HTML pages. Perhaps your software development team is not even responsible for the final page layout. That duty may belong to a dedicated Web page designer in your organization, for example. Your page designers may not have any experience with software development. Fortunately, they don't have to. They can edit the page templates at any point in the product development and maintenance cycle, without ever changing the source code. Thus, WebSnap HTML templates can make server development and maintenance more efficient.

Manipulating server-side script in HTML files

HTML in page templates can be modified at any time in the development cycle. Server-side scripting can be a different matter, however. It is always possible to manipulate the server-side script in the templates outside of the IDE, but it is not recommended for pages generated by an adapter page producer. The adapter page producer is different from ordinary page producers in that it can change the server-side scripting in the page templates at runtime. It can be difficult to predict how your script will act if other script is added dynamically. If you want to manipulate script directly, make sure that your Web page module contains a page producer instead of an adapter page producer.

If you have a Web page module that uses an adapter page producer, you can convert it to use a regular page producer instead by using the following steps.

To modify a Web page module to use a regular page producer

1. You can access the page module view with server-side scripting using the HTML Script tab. In the module you want to convert (let's call it ModuleName), copy all of the information from the HTML Script tab to the ModuleName.html tab, replacing all of the information that it contained previously.

   Note: When the Web Page module uses *TAdapterPageProducer* the page module views become available when this component is double-clicked .

2. Drop a page producer (located on the Internet category of the **Tool Palette**) onto your Web page module.
3. Set the page producer's ScriptEngine property to match that of the adapter page producer it replaces.
4. Change the page producer in the Web page module from the adapter page producer to the new page producer.
5. The adapter page producer has now been bypassed. You may now delete it from the Web page module.

3.2.5.5.1.2 Specifying Application Module Components

Application components provide the Web application's functionality. For example, including an adapter dispatcher component automatically handles HTML form submissions and the return of dynamically generated images. Including a page dispatcher automatically displays the content of a page when the HTTP request pathinfo contains the name of the page.

For information on creating web server applications, see Creating Web Server Applications with WebSnap (see page 2313).

Selecting the Components button on the new WebSnap application dialog displays another dialog that allows you to select one or more of the Web application module components.

The following table contains a brief explanation of the available components:

Web application components

Component type	Description
Application Adapter	Contains information about the application, such as the title. The default type is *TApplicationAdapter*.
End User Adapter	Contains information about the user, such as their name, access rights, and whether they are logged in. The default type is *TEndUserAdapter*. *TEndUserSessionAdapter* may also be selected.
Page Dispatcher	Examines the HTTP request's pathinfo and calls the appropriate page module to return the content of a page. The default type is *TPageDispatcher*.
Adapter Dispatcher	Automatically handles HTML form submissions and requests for dynamic images by calling adapter action and field components. The default type is *TAdapterDispatcher*.
Dispatch Actions	Allows you to define a collection of action items to handle requests based on pathinfo and method type. Action items call user-defined events or request the content of page-producer components. The default type is *TWebDispatcher*.
Locate File Service	Provides control over the loading of template files, and script include files, when the Web application is running. The default type is *TLocateFileService*.
Sessions Service	Stores information about end users that is needed for a short period of time. For example, you can use sessions to keep track of logged-in users and to automatically log a user out after a period of inactivity. The default type is *TSessionsService*.
User List Service	Keeps track of authorized users, their passwords, and their access rights. The default type is *TWebUserList*.

For each of the above components, the component types listed are the default types shipped with the IDE. Users can create their own component types or use third-party component types.

For information about modifying application module components, see Selecting Web Application Module Options (see page 2314).

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.1.3 Creating Web Server Applications with WebSnap

If you look at the source code for WebSnap, you will discover that WebSnap comprises hundreds of objects. In fact, WebSnap is so rich in objects and features that you could spend a long time studying its architecture in detail before understanding it completely. Fortunately, you really don't need to understand the whole WebSnap system before you start developing your server application.

Here you will learn more about how WebSnap works by creating a new Web server application.

To create a new Web server application using the WebSnap architecture:

1. Choose **File** ▶ **New** ▶ **Other**, and select the WebSnap folder from Delphi Projects.
2. In the right pane of the New Items window choose WebSnap Application. A dialog box appears (as shown below)
3. Specify the correct server type (see page 2314).
4. Use the components button to specify application module components (see page 2312).
5. Use the Page Options button to select application module options (see page 2314).

For further information about adding application module components, see Specifying Application Module Components (see page 2312).

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.1.4 Selecting a Server Type

Select one of the following types of Web server application, depending on your application's type of Web server.

Web server application types

Server type	Description
ISAPI and NSAPI	Sets up your project as a DLL with the exported methods expected by the Web server.
Apache	Sets up your project as a DLL with the exported methods expected by the appropriate Apache Web server. Both Apache 1 and 2 are supported.
CGI stand-alone	Sets up your project as a console application which conforms to the Common Gateway Interface (CGI) standard.
Web App Debugger executable	Creates an environment for developing and testing Web server applications. This type of application is not intended for deployment.

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.1.5 Selecting Web Application Module Options

If the selected application module type is a page module, you can associate a name with the page by entering a name in the Page Name field in the New WebSnap Application dialog box. At runtime, the instance of this module can be either kept in cache or removed from memory when the request has been serviced. Select either of the options from the Caching field. You can select more page module options by choosing the Page Options button.

For information on adding application module components, see Specifying Application Module Components (see page 2312).

The Application Module Page Options dialog is displayed and provides the following categories:

Note: The *AdapterPageProducer* supports only JScript.

- Producer: The producer type for the page can be set to one of *AdapterPageProducer*, *DataSetPageProducer*, *InetXPageProducer*, *PageProducer*, or *XSLPageProducer*. If the selected page producer supports scripting, then use the Script Engine drop-down list to select the language used to script the page.
- HTML: When the selected producer uses an HTML template this group will be visible.
- XSL: When the selected producer uses an XSL template, such as *TXSLPageProducer*, this group will be visible.
- New File: Check New File if you want a template file to be created and managed as part of the unit. A managed template file appears in the **Project Manager** and has the same file name and location as the unit source file. Uncheck New File if you want to use the properties of the producer component (typically the *HTMLDoc* or *HTMLFile* property).
- Template: When New File is checked, choose the default content for the template file from the Template drop-down. The standard template displays the title of the application, the title of the page, and hyperlinks to published pages. The blank template creates a blank page.
- Page: Enter a page name and title for the page module. The page name is used to reference the page in an HTTP request or within the application's logic, whereas the title is the name that the end user will see when the page is displayed in a browser.

- Published: Check Published to allow the page to automatically respond to HTTP requests where the page name matches the pathinfo in the request message.
- Login Required: Check Login Required to require the user to log on before the page can be accessed.

See Also

Type of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.2 Creating Web Server Applications Using WebSnap

WebSnap augments Web Broker with additional components, wizards, and views—making it easier to build Web server applications that deliver complex, data-driven Web pages. WebSnap's support for multiple modules and for server-side scripting makes development and maintenance easier for teams of developers and Web designers.

WebSnap allows HTML design experts on your team to make a more effective contribution to Web server development and maintenance. The final product of the WebSnap development process includes a series of scriptable HTML page templates. These pages can be changed using HTML editors that support embedded script tags, like Microsoft FrontPage, or even a simple text editor. Changes can be made to the templates as needed, even after the application is deployed. There is no need to modify the project source code at all, which saves valuable development time. Also, WebSnap's multiple module support can be used to partition your application into smaller pieces during the coding phases of your project. Developers can work more independently.

The dispatcher components automatically handle requests for page content, HTML form submissions, and requests for dynamic images. WebSnap components called adapters provide a means to define a scriptable interface to the business rules of your application. For example, the *TDataSetAdapter* object is used to make dataset components scriptable. You can use WebSnap producer components to quickly build complex, data-driven forms and tables, or to use XSL to generate a page. You can use the session component to keep track of end users. You can use the user list component to provide access to user names, passwords, and access rights.

The Web application wizard allows you to quickly build an application that is customized with the components that you will need. The Web page module wizard allows you to create a module that defines a new page in your application. Or use the Web data module wizard to create a container for components that are shared across your Web application.

When the Web Page module uses *TAdapterPageProducer* the page module views become available when this component is double-clicked . The page module views show the result of server-side scripting without running the application. You can view the generated HTML in text form using the HTML Result tab. The HTML Script tab shows the page with server-side scripting, which is used to generate HTML for the page.

The following topics explain how to use the WebSnap components to create a Web server application:

- Fundamental WebSnap components (see page 2326)
- Creating Web Server Applications (see page 2313)
- Server-side scripting in WebSnap (see page 2333)
- Dispatching requests (see page 2320)

Note: WebSnap is being deprecated in RAD Studio. Although WebSnap is still documented in the online help, the WebSnap product is no longer fully supported. As an alternative, you should begin using IntraWeb (VCL for the Web). IntraWeb (see page 2254) is documented in this online help. For more documentation on VCL for the Web, go to http://www.atozed.com/intraweb/docs/.

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

Using IntraWeb (see page 2254)

3.2.5.5.3 Dispatching requests and responses

Topics

Name	Description
Adapter dispatcher operation (see page 2316)	
Dispatcher Components (see page 2319)	The dispatcher components in the Web application module control the flow of the application. The dispatchers determine how to handle certain types of HTTP request messages by examining the HTTP request. The adapter dispatcher component (*TAdapterDispatcher*) looks for a content field, or a query field, that identifies an adapter action component or an adapter image field component. If the adapter dispatcher finds a component, it passes control to that component. The Web dispatcher component (*TWebDispatcher*) maintains a collection of action items (of type *TWebActionItem*) that know how to handle certain types of HTTP request messages.... more (see page 2319)
Dispatching Action Items (see page 2320)	When responding to a request, the Web dispatcher (*TWebDispatcher*) searches through its list of action items for one that: • matches the PathInfo portion of the target URL's request message, and • can provide the service specified as the method of the request message. It accomplishes this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item. When the dispatcher finds the appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following: • Fills in the response content... more (see page 2320)
Dispatching Requests and Responses (see page 2320)	One reason to use WebSnap for your Web server application development is that WebSnap components automatically handle HTML requests and responses. Instead of writing event handlers for common page transfer chores, you can focus your efforts on your business logic and server design. Still, it can be helpful to understand how WebSnap applications handle HTML requests and responses. This section gives you an overview of that process. Before handling any requests, the Web application module initializes the Web context object (of type TWebContext). The Web context object, which is accessed by calling the global WebContext function, provides global access to... more (see page 2320)
Page dispatcher operation (see page 2321)	When the page dispatcher receives a client request, it determines the page name by checking the PathInfo portion of the target URL's request message. If the PathInfo portion is not blank, the page dispatcher uses the ending word of PathInfo as the page name. If the PathInfo portion is blank, the page dispatcher tries to determine a default page name. If the page dispatcher's *DefaultPage* property contains a page name, the page dispatcher uses this name as the default page name. If the *DefaultPage* property is blank and the Web application module is a page module, the page dispatcher uses... more (see page 2321)

3.2.5.5.3.1 Adapter dispatcher operation

Topics

Name	Description
Adapter Dispatcher Operation (see page 2317)	The adapter dispatcher component (*TAdapterDispatcher*) automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components.

Receiving Adapter Requests and Generating Responses (see page 2318)	When the adapter dispatcher receives a client request, the adapter dispatcher creates adapter request and adapter response objects to hold information about that HTTP request. The adapter request and adapter response objects are stored in the Web context to allow access during the processing of the request. The adapter dispatcher creates two types of adapter request objects: action and image. It creates the action request object when executing an adapter action. It creates the image request object when retrieving an image from an adapter field. The adapter response object is used by the adapter component to indicate the response to... more (see page 2318)

3.2.5.5.3.1.1 Adapter Dispatcher Operation

The adapter dispatcher component (*TAdapterDispatcher*) automatically handles HTML form submissions, and requests for dynamic images, by calling adapter action and field components.

Using adapter components to generate content

For WebSnap applications to automatically execute adapter actions and retrieve dynamic images from adapter fields, the HTML content must be properly constructed. If the HTML content is not properly constructed, the resulting HTTP request will not contain the information that the adapter dispatcher needs to call adapter action and field components.

To reduce errors in constructing the HTML page, adapter components indicate the names and values of HTML elements. Adapter components have methods that retrieve the names and values of hidden fields that must appear on an HTML form designed to update adapter fields. Typically, page producers use server-side scripting to retrieve names and values from adapter components and then uses this information to generate HTML. For example, the following script constructs an <IMG> element that references the field called Graphic from Adapter1:

```
<img src="<%=Adapter1.Graphic.Image.AsHREF%>" alt="<%=Adapter1.Graphic.DisplayText%>">
```

When the Web application evaluates the script, the HTML src attribute will contain the information necessary to identify the field and any parameters that the field component needs to retrieve the image. The resulting HTML might look like this:

```
<img src="?_lSpecies No=90090&__id=DM.Adapter1.Graphic" alt="(GRAPHIC)">
```

When the browser sends an HTTP request to retrieve this image to the Web application, the adapter dispatcher will be able to determine that the Graphic field of Adapter1, in the module DM, should be called with "Species No=90090" as a parameter. The adapter dispatcher will call the Graphic field to write an appropriate HTTP response.

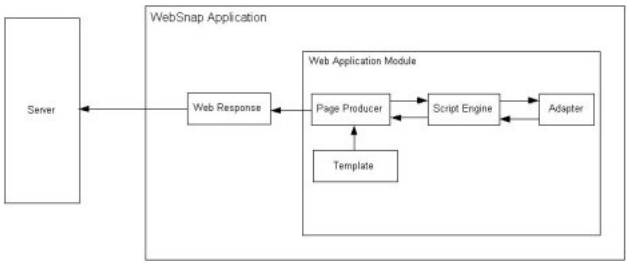
The following script constructs an <A> element referencing the EditRow action of Adapter1 and creates a hyperlink to a page called Details:

```
<a href="<%=Adapter1.EditRow.LinkToPage("Details", Page.Name).AsHREF%>">Edit...</a>
```

The resulting HTML might look like this:

```
<a href="?&_lSpecies No=90310&__sp=Edit&__fp=Grid&__id=DM.Adapter1.EditRow">Edit...</a>
```

The end user clicks this hyperlink, and the browser sends an HTTP request. The adapter dispatcher can determine that the EditRow action of Adapter1, in the module DM, should be called with the parameter Species No=903010. The adapter dispatcher also displays the Edit page if the action executes successfully, and displays the Grid page if action execution fails. It then calls the EditRow action to locate the row to be edited, and the page named Edit is called to generate an HTTP response. The following figure shows how adapter components are used to generate content.



See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.3.1.2 Receiving Adapter Requests and Generating Responses

When the adapter dispatcher receives a client request, the adapter dispatcher creates adapter request and adapter response objects to hold information about that HTTP request. The adapter request and adapter response objects are stored in the Web context to allow access during the processing of the request.

The adapter dispatcher creates two types of adapter request objects: action and image. It creates the action request object when executing an adapter action. It creates the image request object when retrieving an image from an adapter field.

The adapter response object is used by the adapter component to indicate the response to an adapter action or adapter image request. There are two types of adapter response objects, action and image.

Action requests

Action request objects are responsible for breaking the HTTP request down into information needed to execute an adapter action. The types of information needed for executing an adapter action may include the following request information:

Request information found in action requests

Request informaton	Description
Component name	Identifies the adapter action component.
Adapter mode	Defines a mode. For example, TDataSetAdapter supports Edit, Insert, and Browse modes. An adapter action may execute differently depending on the mode.
Success page	Identifies the page displayed after successful execution of the action.
Failure page	Identifies the page displayed if an error occurs during execution of the action.
Action request parameters	Identifies the parameters need by the adapter action. For example, the TDataSetAdapter Apply action will include the key values identifying the record to be updated.
Adapter field values	Specifies values for the adapter fields passed in the HTTP request when an HTML form is submitted. A field value can include new values entered by the end user, the original values of the adapter field, and uploaded files.
Record keys	Specifies keys that uniquely identify each record.

Generating action responses

Action response objects generate an HTTP response on behalf of an adapter action component. The adapter action indicates the type of response by setting properties within the object, or by calling methods in the action response object. The properties include:

- *RedirectOptions*—The redirect options indicate whether to perform an HTTP redirect instead of returning HTML content.
- *ExecutionStatus*—Setting the status to success causes the default action response to be the content of the success page identified in the Action Request.

The action response methods include:

- *RespondWithPage*—The adapter action calls this method when a particular Web page module should generate the response.
- *RespondWithComponent*—The adapter action calls this method when the response should come from the Web page module containing this component.
- *RespondWithURL*—The adapter action calls this method when the response is a redirect to a specified URL.

When responding with a page, the action response object attempts to use the page dispatcher to generate page content. If it does not find the page dispatcher, it calls the Web page module directly.

The following figure illustrates how action request and action response objects handle a request.

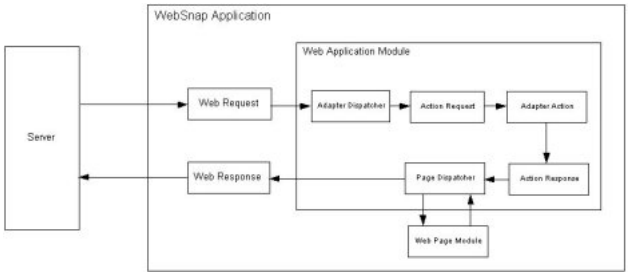


Image request

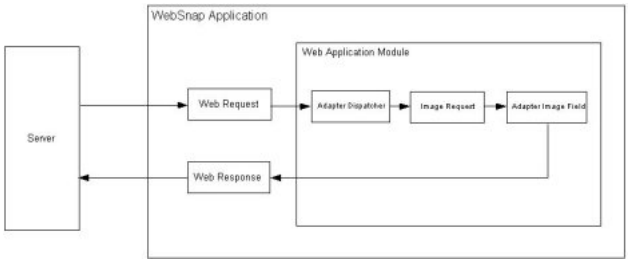
The image request object is responsible for breaking the HTTP request down into the information required by the adapter image field to generate an image. The types of information represented by the Image Request include:

- Component name - Identifies the adapter field component.
- Image request parameters - Identifies the parameters needed by the adapter image. For example, the *TDataSetAdapterImageField* object needs key values to identify the record that contains the image.

Image response

The image response object contains the *TWebResponse* object. Adapter fields respond to an adapter request by writing an image to the Web response object.

The following figure illustrates how adapter image fields respond to a request.



See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.3.2 Dispatcher Components

The dispatcher components in the Web application module control the flow of the application. The dispatchers determine how to handle certain types of HTTP request messages by examining the HTTP request.

The adapter dispatcher component (*TAdapterDispatcher*) looks for a content field, or a query field, that identifies an adapter action component or an adapter image field component. If the adapter dispatcher finds a component, it passes control to that component.

The Web dispatcher component (*TWebDispatcher*) maintains a collection of action items (of type *TWebActionItem*) that know how to handle certain types of HTTP request messages. The Web dispatcher looks for an action item that matches the request. If it finds one, it passes control to that action item. The Web dispatcher also looks for auto-dispatching components that can handle the request.

The page dispatcher component (*TPageDispatcher*) examines the *PathInfo* property of the *TWebRequest* object, looking for the

name of a registered Web page module. If the dispatcher finds a Web page module name, it passes control to that module.

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.3.3 Dispatching Action Items

When responding to a request, the Web dispatcher (*TWebDispatcher*) searches through its list of action items for one that:

- matches the PathInfo portion of the target URL's request message, and
- can provide the service specified as the method of the request message.

It accomplishes this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds the appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response, or signals that the request has been completely handled.
- Adds to the response, and then allows other action items to complete the job.
- Defers the request to other action items.

After the dispatcher has checked all of its action items, if the message was not handled correctly, the dispatcher checks for specially registered auto-dispatching components that do not use action items. (These components are specific to multi-tiered database applications.) If the request message is still not fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.3.4 Dispatching Requests and Responses

One reason to use WebSnap for your Web server application development is that WebSnap components automatically handle HTML requests and responses. Instead of writing event handlers for common page transfer chores, you can focus your efforts on your business logic and server design. Still, it can be helpful to understand how WebSnap applications handle HTML requests and responses. This section gives you an overview of that process.

Before handling any requests, the Web application module initializes the Web context object (of type TWebContext). The Web context object, which is accessed by calling the global WebContext function, provides global access to variables used by components servicing the request. For example, the Web context contains the TWebRequest and TWebResponse objects to represent the HTTP request message and the response that should be returned.

The following topics describe Web request handling:

- Using dispatcher components (see page 2319)
- Adapter dispatcher operation (see page 2317)
- Dispatching action items (see page 2320)
- Page dispatcher operation (see page 2321)

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

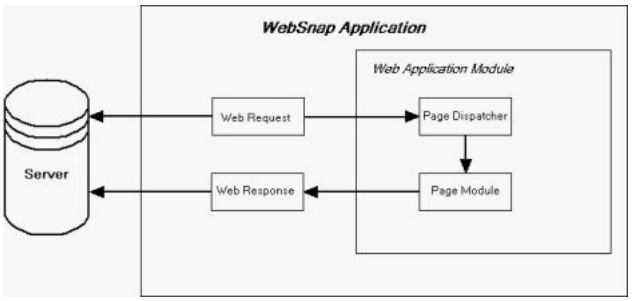
3.2.5.5.3.5 Page dispatcher operation

When the page dispatcher receives a client request, it determines the page name by checking the PathInfo portion of the target URL's request message. If the PathInfo portion is not blank, the page dispatcher uses the ending word of PathInfo as the page name. If the PathInfo portion is blank, the page dispatcher tries to determine a default page name.

If the page dispatcher's *DefaultPage* property contains a page name, the page dispatcher uses this name as the default page name. If the *DefaultPage* property is blank and the Web application module is a page module, the page dispatcher uses the name of the Web application module as the default page name.

If the page name is not blank, the page dispatcher searches for a Web page module with a matching name. If it finds a Web page module, it calls that module to generate a response. If the page name is blank, or if the page dispatcher does not find a Web page module, the page dispatcher raises an exception.

The following figure shows how the page dispatcher responds to a request.



See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.4 Fundamental WebSnap components

Topics

Name	Description
Web modules (see page 2322)	
Adapters (see page 2325)	Adapters define a script interface to your server application. They allow you to insert scripting languages into a page and retrieve information by making calls from your script code to the adapters. For example, you can use an adapter to define data fields to be displayed on an HTML page. A scripted HTML page can then contain HTML content and script statements that retrieve the values of those data fields. This is similar to the transparent tags used in Web Broker applications. Adapters also support actions that execute commands. For example, clicking on a hyperlink or submitting an HTML form... more (see page 2325)

Fundamental WebSnap Components (see page 2326)	Before you can build Web server applications using WebSnap, you must first understand the fundamental components used in WebSnap development. They fall into three categories: • Web modules (see page 2324), which contain the components that make up the application and define pages • Adapters (see page 2325), which provide an interface between HTML pages and the Web server application itself • Page producers (see page 2326), which contain the routines that create the HTML pages to be served to the end user The following sections examine each type of component in more detail.
Page Producers (see page 2326)	Page producers to generate content on behalf of a Web page module. Page producers provide the following functionality: • They generate HTML content. • They can reference an external file using the HTMLFile property, or the internal string using the HTMLDoc property. • When the producers are used with a Web page module, the template can be a file associated with a unit. • Producers dynamically generate HTML that can be inserted into the template using transparent tags or active scripting. Transparent tags can be used in the same way as WebBroker applications. To learn more about using transparent tags, see Converting HTML-transparent (see page 2269)... more (see page 2326)

3.2.5.5.4.1 Web modules

Topics

Name	Description
Web Application Module Types (see page 2323)	Web application modules provide centralized control for business rules and non-visual components in the Web application. The two types of Web application modules are tabulated below. ***Web application module types***
Web Data Modules (see page 2324)	Like standard data modules, Web data modules are a container for components from the palette. Data modules provide a design surface for adding, removing, and selecting components. The Web data module differs from a standard data module in the structure of the unit and the interfaces that the Web data module implements. Use the Web data module as a container for components that are shared across your application. For example, you can put a dataset component in a data module and access the dataset from both: • a page module that displays a grid, and • a page module that displays an... more (see page 2324)

Web Modules (see page 2324)	Web modules are the basic building block of WebSnap applications. Every WebSnap server application must have at least one Web module. More can be added as needed. There are four Web module types: • Web application page modules (TWebAppPageModule objects) • *Web application data modules* (TWebAppDataModule objects) • *Web page modules (*TWebPageModule objects) • *Web data modules (*TWebDataModule objects) Web page modules and Web application page modules provide content for Web pages. Web data modules and Web application data modules act as containers for components shared across your application; they serve the same purpose in WebSnap applications that ordinary data modules serve... more (see page 2324)
Web Page Modules (see page 2325)	Each Web page module has a page producer associated with it. When a request is received, the page dispatcher analyzes the request and calls the appropriate page module to process the request and return the content of the page. Like Web data modules, Web page modules are containers for components. A Web page module is more than a mere container, however. A Web page module is used specifically to produce a Web page.

3.2.5.5.4.1.1 Web Application Module Types

Web application modules provide centralized control for business rules and non-visual components in the Web application. The two types of Web application modules are tabulated below.

Web application module types

Web application module type	Description
Page	Creates a content page. The page module contains a page producer which is responsible for generating the content of a page. The page producer displays its associated page when the HTTP request pathinfo matches the page name. The page can act as the default page when the pathinfo is blank.
Data	Used as a container for components shared by other modules, such as database components used by multiple Web page modules.

Web application modules act as containers for components that perform functions for the application as a whole—such as dispatching requests, managing sessions, and maintaining user lists. If you are already familiar with the Web Broker architecture, you can think of Web application modules as being similar to *TWebApplication* objects. Web application modules also contain the functionality of a regular Web module, either page or data, depending on the Web application module type. Your project can contain only one Web application module. You will never need more than one anyway; you can add regular Web modules to your server to provide whatever extra features you want.

Use the Web application module to contain the most basic features of your server application. If your server will maintain a home page of some sort, you may want to make your Web application module a *TWebAppPageModule* instead of a *TWebAppDataModule*, so you don't have to create an extra Web page module for that page.

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.4.1.2 Web Data Modules

Like standard data modules, Web data modules are a container for components from the palette. Data modules provide a design surface for adding, removing, and selecting components. The Web data module differs from a standard data module in the structure of the unit and the interfaces that the Web data module implements.

Use the Web data module as a container for components that are shared across your application. For example, you can put a dataset component in a data module and access the dataset from both:

- a page module that displays a grid, and
- a page module that displays an input form.

You can also use Web data modules to contain sets of components that can be used by several different Web server applications.

Structure of a Web data module unit

Standard data modules have a variable called a form variable, which is used to access the data module object. Web data modules replace the variable with a function, which is defined in a Web data module's unit and has the same name as the Web data module. The function's purpose is the same as that of the variable it replaces. WebSnap applications may be multi-threaded and may have multiple instances of a particular module to service multiple requests concurrently. Therefore, the function is used to return the correct instance.

The Web data module unit also registers a factory to specify how the module should be managed by the WebSnap application. For example, flags indicate whether to cache the module and reuse it for other requests or to destroy the module after a request has been serviced.

See Also

The Web Application Object (see page 2288)

Using Data Modules and Remote Data Modules (see page 1912)

Web Modules (see page 2324)

3.2.5.5.4.1.3 Web Modules

Web modules are the basic building block of WebSnap applications. Every WebSnap server application must have at least one Web module. More can be added as needed. There are four Web module types:

- Web application page modules (TWebAppPageModule objects)
- *Web application data modules* (TWebAppDataModule objects)
- *Web page modules (*TWebPageModule objects)
- *Web data modules (*TWebDataModule objects)

Web page modules and Web application page modules provide content for Web pages. Web data modules and Web application data modules act as containers for components shared across your application; they serve the same purpose in WebSnap applications that ordinary data modules serve in regular applications. You can include any number of Web page or data modules in your server application.

You may be wondering how many Web modules your application needs. Every WebSnap application needs one (and only one) Web application module of some type. Beyond that, you can add as many Web page or data modules as you need.

For Web page modules, a good rule of thumb is one per page style. If you intend to implement a page that can use the format of an existing page, you may not need a new Web page module. Modifications to an existing page module may suffice. If the page is very different from your existing modules, you will probably want to create a new module. For example, let's say you are trying to build a server to handle online catalog sales. Pages which describe available products might all share the same Web page module, since the pages can all contain the same basic information types using the same layout. An order form, however, would probably require a different Web page module, since the format and function of an order form is different from that of an item description page.

The rules are different for Web data modules. Components that can be shared by many different Web modules should be placed in a Web data module to simplify shared access. You will also want to place components that can be used by many different Web applications in their own Web data module. That way you can easily share those items among applications. Of course, if neither of these circumstances applies you might choose not to use Web data modules at all. Use them the same way you would use regular data modules, and let your own judgment and experience be your guide.

The following topics describe Web modules in greater detail:

- Web application module types (see page 2323)
- Web page modules (see page 2325)
- Web data modules (see page 2324)

See Also

The Web Application Object (see page 2288)

Using Data Modules and Remote Data Modules (see page 1912)

3.2.5.5.4.1.4 Web Page Modules

Each Web page module has a page producer associated with it. When a request is received, the page dispatcher analyzes the request and calls the appropriate page module to process the request and return the content of the page.

Like Web data modules, Web page modules are containers for components. A Web page module is more than a mere container, however. A Web page module is used specifically to produce a Web page.

Page producer component

Web page modules have a property that identifies the page producer (see page 2326) component responsible for generating content for the page. The WebSnap page module wizard automatically adds a producer when creating a Web page module. You can change the page producer component later by dropping in a different producer from the WebSnap category . However, if the page module has a template file, be sure that the content of this file is compatible with the replacement producer component.

Page name

Web page modules have a page name that can be used to reference the page in an HTTP request or within the application's logic. A factory in the Web page module's unit specifies the page name for the Web page module.

Producer template

Most page producers use a template. HTML templates typically contain some static HTML mixed in with transparent tags or server-side scripting. When page producers create their content, they replace the transparent tags with appropriate values and execute the server-side script to produce the HTML that is displayed by a client browser. (The XSLPageProducer is an exception to this. It uses XSL templates, which contain XSL rather than HTML. The XSL templates do not support transparent tags or server-side script.)

Web page modules may have an associated template file that is managed as part of the unit. A managed template file appears in the **Project Manager** and has the same base file name and location as the unit service file. If the Web page module does not have an associated template file, the properties of the page producer component specify the template.

See Also

The Web Application Object (see page 2288)

Web Modules (see page 2324)

3.2.5.5.4.2 Adapters

Adapters define a script interface to your server application. They allow you to insert scripting languages into a page and retrieve information by making calls from your script code to the adapters. For example, you can use an adapter to define data fields to be displayed on an HTML page. A scripted HTML page can then contain HTML content and script statements that retrieve the

values of those data fields. This is similar to the transparent tags used in Web Broker applications. Adapters also support actions that execute commands. For example, clicking on a hyperlink or submitting an HTML form can initiate adapter actions.

Adapters simplify the task of creating HTML pages dynamically. By using adapters in your application, you can include object-oriented script that supports conditional logic and looping. Without adapters and server-side scripting, you must write more of your HTML generation logic in event handlers. Using adapters can significantly reduce development time.

See Server-side scripting in WebSnap (see page 2333) for more details about scripting.

Four types of adapter components can be used to create page content: fields, actions, errors and records.

Fields

Fields are components that the page producer uses to retrieve data from your application and to display the content on a Web page. Fields can also be used to retrieve an image. In this case, the field returns the address of the image written to the Web page. When a page displays its content, a request is sent to the Web server application, which invokes the adapter dispatcher to retrieve the actual image from the field component.

Actions

Actions are components that execute commands on behalf of the adapter. When a page producer generates its page, the scripting language calls adapter action components to return the name of the action along with any parameters necessary to execute the command. For example, consider clicking a button on an HTML form to delete a row from a table. This returns, in the HTTP request, the action name associated with the button and a parameter indicating the row number. The adapter dispatcher locates the named action component and passes the row number as a parameter to the action.

Errors

Adapters keep a list of errors that occur while executing an action. Page producers can access this list of errors and display them in the Web page that the application returns to the end user.

Records

Some adapter components, such as *TDataSetAdapter*, represent multiple records. The adapter provides a scripting interface which allows iteration through the records. Some adapters support paging and iterate only through the records on the current page.

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.4.3 Fundamental WebSnap Components

Before you can build Web server applications using WebSnap, you must first understand the fundamental components used in WebSnap development. They fall into three categories:

- Web modules (see page 2324), which contain the components that make up the application and define pages
- Adapters (see page 2325), which provide an interface between HTML pages and the Web server application itself
- Page producers (see page 2326), which contain the routines that create the HTML pages to be served to the end user

The following sections examine each type of component in more detail.

See Also

The Web Application Object (see page 2288)

3.2.5.5.4.4 Page Producers

Page producers to generate content on behalf of a Web page module. Page producers provide the following functionality:

- They generate HTML content.
- They can reference an external file using the HTMLFile property, or the internal string using the HTMLDoc property.
- When the producers are used with a Web page module, the template can be a file associated with a unit.
- Producers dynamically generate HTML that can be inserted into the template using transparent tags or active scripting. Transparent tags can be used in the same way as WebBroker applications. To learn more about using transparent tags, see Converting HTML-transparent tags (see page 2269). Active scripting support allows you to embed JScript or VBScript inside the HTML page.

The standard WebSnap method for using page producers is as follows. When you create a Web page module, you must choose a page producer type in the Web Page Module wizard. You have many choices, but most WebSnap developers prototype their pages by using an adapter page producer, TAdapterPageProducer. The adapter page producer lets you build a prototype Web page using a process analogous to the standard component model. You add a type of form, an adapter form, to the adapter page producer. As you need them, you can add adapter components (such as adapter grids) to the adapter form. Using adapter page producers, you can create Web pages in a way that is similar to the standard technique for building user interfaces.

There are some circumstances where switching from an adapter page producer to a regular page producer is more appropriate. For example, part of the function of an adapter page producer is to dynamically generate script in a page template at runtime. You may decide that static script would help optimize your server. Also, users who are experienced with script may want to make changes to the script directly. In this case, a regular page producer must be used to avoid conflicts between dynamic and static script. To learn how to change to a regular page producer, see the Advanced HTML design (see page 2311) topic.

You can also use page producers the same way you would use them in Web Broker applications, by associating the producer with a Web dispatcher action item. The advantages of using the Web page module are

- the ability to preview the page's layout without running the application, and
- the ability to associate a page name with the module, so that the page dispatcher can call the page producer automatically.

See Also

Creating Web Server Applications with WebSnap (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.5 Login support

Topics

Name	Description
Adding Login Support (see page 2328)	In order to implement login support, you need to make sure your Web application module has the following components: • A user list service (an object of type TWebUserList), which contains the usernames, passwords and permissions for server users • A sessions service (TSessionsService), which stores information about users currently logged in to the server • An end user adapter (TEndUserSessionAdapter) which handles actions associated with logging in When you first create your Web server application, you can add these components using the New WebSnap Application dialog box. Click the Components button on that dialog to display the New Web App Components dialog... more (see page 2328)

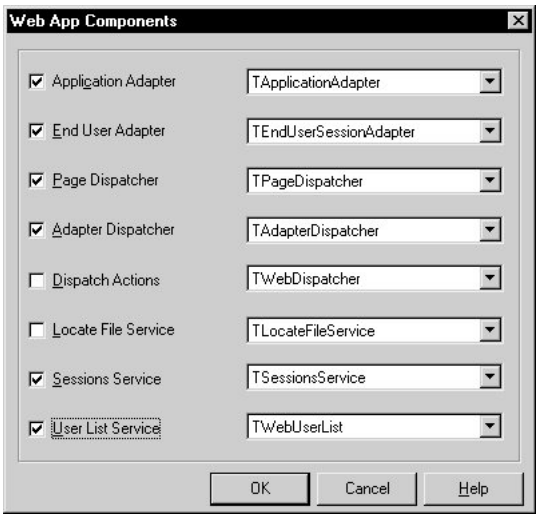
Login Pages (see page 2329)	Of course, your Websnap application also needs a login page. Users enter their username and password for authentication, either while trying to access a restricted page or prior to such an attempt. The user can also specify which page they receive when authentication is completed. If the username and password match a user in the Web user list, the user acquires the appropriate access rights and is forwarded to the page specified on the login page. If the user isn't authenticated, the login page may be redisplayed (the default action) or some other action may occur. Fortunately, WebSnap makes it... more (see page 2329)
Login Support (see page 2330)	Many Web server applications require login support. For example, a server application may require a user to login before granting access to some parts of a Web site. Pages may have a different appearance for different users; logins may be necessary to enable the Web server to send the right pages. Also, because servers have physical limitations on memory and processor cycles, server applications sometimes need the ability to limit the number of users at any given time. With WebSnap, incorporating login support into your Web server application is fairly simple and straightforward. You can add login support, either by... more (see page 2330)
Setting Pages to Require Logins (see page 2330)	Once you have a working login page, you must require logins for those pages which need controlled access. The easiest way to have a page require logins is to design that requirement into the page. When you first create a Web page module, check the Login Required box in the Page section of the New WebSnap Page Module dialog box. If you create a page without requiring logins, you can change your mind later.
User Access Rights (see page 2331)	User access rights are an important part of any Web server application. You need to be able to control who can view and modify the information your server provides. For example, let's say you are building a server application to handle online retail sales. It makes sense to allow users to view items in your catalog, but you don't want them to be able to change your prices! Clearly, access rights are an important issue. Fortunately, WebSnap offers you several ways to control access to pages and server content. In previous topics, you saw how you can control page access... more (see page 2331)
Using the Sessions Service (see page 2332)	The sessions service, which is an object of type TSessionsService, keeps track of the users who are logged into your Web server application. The sessions service is responsible for assigning a different session for each user and for associating name/value pairs (such as a username) with a user. Information contained in a sessions service is stored in the application's memory. Therefore, the Web server application must keep running between requests for the sessions service to work. Some server application types, such as CGI, terminate between requests. **Note:** If you want your application to support logins, be sure to use a... more (see page 2332)

3.2.5.5.5.1 Adding Login Support

In order to implement login support, you need to make sure your Web application module has the following components:

- A user list service (an object of type TWebUserList), which contains the usernames, passwords and permissions for server users
- A sessions service (TSessionsService), which stores information about users currently logged in to the server
- An end user adapter (TEndUserSessionAdapter) which handles actions associated with logging in

When you first create your Web server application, you can add these components using the New WebSnap Application dialog box. Click the Components button on that dialog to display the New Web App Components dialog box. Check the End User Adapter, Sessions Service and Web User List boxes. Select *TEndUserSessionAdapter* on the drop down menu next to the End User Adapter box to select the end user adapter type. (The default choice, TEndUserSessionAdapter, is not appropriate for login support because it cannot track the current user.) When you're finished, your dialog should look like the one shown below. Click OK twice to dismiss the dialog boxes. Your Web application module now has the necessary components for login support.



If you are adding login support to an existing Web application module, you can drag these components directly into your module from the WebSnap category of the **Tool Palette**. The Web application module will configure itself automatically.

The sessions service and the end user adapter may not require your attention during your design phase, but the Web user list probably will. You can add default users and set their read/modify permissions through the WebUserList component editor. Double-click on the component to display an editor which lets you set usernames, passwords and access rights. For more information on how to set up access rights, see the topic "User access rights" (see page 2331).

For information on login support, see Login Support (see page 2330).

3.2.5.5.5.2 Login Pages

Of course, your Websnap application also needs a login page. Users enter their username and password for authentication, either while trying to access a restricted page or prior to such an attempt. The user can also specify which page they receive when authentication is completed. If the username and password match a user in the Web user list, the user acquires the appropriate access rights and is forwarded to the page specified on the login page. If the user isn't authenticated, the login page may be redisplayed (the default action) or some other action may occur.

Fortunately, WebSnap makes it easy to create a simple login page using a Web page module and the adapter page producer. To create a login page, start by creating a new Web page module. Choose **File ▸ New ▸ Other,** and select WebSnap from the Delphi Projects folder. In the right pane of the New Items window select the WebSnap Page Module. Select AdapterPageProducer as the page producer type. Fill in the other options however you like. Login tends to be a good name for the login page.

Now you should add the most basic login page fields: a username field, a password field, a selection box for selecting which page the user receives after logging in, and a Login button which submits the page and authenticates the user.

To add these fields:

1. Add a TLoginFormAdapter component (which you can find on the WebSnap category of the **Tool Palette**) to the Web page module you just created.
2. Double-click the *AdapterPageProducer* component to display a Web page editor window.
3. Right-click the *AdapterPageProducer* in the top left pane and select New Component. In the Add Web Component dialog box, select *AdapterForm* and click OK.
4. Add an *AdapterFieldGroup* to the AdapterForm. (Right-click the *AdapterForm* in the top left pane and select New Component. In the Add Web Component dialog box, select *AdapterFieldGroup* and click OK.)
5. Now go to the **Object Inspector** and set the *Adapter* property of your *AdapterFieldGroup* to your *LoginFormAdapter*. The UserName, Password and NextPage fields should appear automatically in the Browser tab of the Web page editor (accessed by double clicking the *AdapterPageProducer*) .

So, WebSnap takes care of most of the work in a few simple steps. The login page is still missing a Login button, which submits the information on the form for authentication.

To add a Login button:

1. Add an *AdapterCommandGroup* to the *AdapterForm*.
2. Add an *AdapterActionButton* to the *AdapterCommandGroup*. Change its *DisplayComponent* to AdapterFieldGroup using the **Object Inspector**.
3. Click on the *AdapterActionButton* (listed in the upper right pane of the Web page editor) and change its *ActionName* property to Login using the **Object Inspector**. You can see a preview of your login page in the Web page editor's Browser tab.

If the button doesn't appear below the *AdapterFieldGroup*, make sure that the *AdapterCommandGroup* is listed below the *AdapterFieldGroup* on the Web page editor. If it appears above, select the *AdapterCommandGroup* and click the down arrow on the Web page editor. (In general, Web page elements appear vertically in the same order as they appear in the Web page editor.)

There is one more step necessary before your login page becomes functional. You need to specify which of your pages is the login page in your end user session adapter. To do so, select the *EndUserSessionAdapter* component in your Web application module. In the **Object Inspector**, change the *LoginPage* property to the name of your login page. Your login page is now enabled for all the pages in your Web server application.

3.2.5.5.5.3 Login Support

Many Web server applications require login support. For example, a server application may require a user to login before granting access to some parts of a Web site. Pages may have a different appearance for different users; logins may be necessary to enable the Web server to send the right pages. Also, because servers have physical limitations on memory and processor cycles, server applications sometimes need the ability to limit the number of users at any given time.

With WebSnap, incorporating login support into your Web server application is fairly simple and straightforward. You can add login support, either by designing it in from the beginning of your development process or by retrofitting it onto an existing application.

For additional information on adding login support, refer to Adding Login Support (see page 2328).

3.2.5.5.5.4 Setting Pages to Require Logins

Once you have a working login page, you must require logins for those pages which need controlled access. The easiest way to have a page require logins is to design that requirement into the page. When you first create a Web page module, check the Login Required box in the Page section of the New WebSnap Page Module dialog box.

If you create a page without requiring logins, you can change your mind later.

To require logins after a Web page module has been created:

1. Open the source code file associated with the Web page module in the editor.
2. Scroll down to the implementation section. In the parameters for the WebRequestHandler.AddWebModuleFactory command, find the creator of the TWebPageInfo object. It should look like this:

```
TWebPageInfo.Create([wpPublished {, wpLoginRequired}], '.html')
```

3. Uncomment the *wpLoginRequired* portion of the parameter list by removing the curly braces. The TWebPageInfo creator should now look like this:

```
TWebPageInfo.Create([wpPublished , wpLoginRequired], '.html')
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished << wpLoginRequired, ".html", "", "", "", "");
```

To remove the login requirement from a page, reverse the process and recomment the *wpLoginRequired* portion of the creator.

Note: You can use the same process to make the page published or not. Simply add or remove comment marks around the *wpPublished* portion as needed.

3.2.5.5.5.5 User Access Rights

User access rights are an important part of any Web server application. You need to be able to control who can view and modify the information your server provides. For example, let's say you are building a server application to handle online retail sales. It makes sense to allow users to view items in your catalog, but you don't want them to be able to change your prices! Clearly, access rights are an important issue.

Fortunately, WebSnap offers you several ways to control access to pages and server content. In previous topics, you saw how you can control page access by requiring logins. You have other options as well. For example:

- You can show data fields in an edit box to users with appropriate modify access rights; other users will see the field contents, but not have the ability to edit them.
- You can hide specific fields from users who don't have the correct view access rights.
- You can prevent unauthorized users from receiving specific pages.

Descriptions for implementing these behaviors are included in this topic.

Dynamically displaying fields as edit or text boxes

If you use the adapter page producer, you can change the appearance of page elements for users with different access rights. For example, the Biolife demo (found in the WebSnap subdirectory of the Demos directory) contains a form page which shows all the information for a given species. The form appears when the user clicks a Details button on the grid. A user logged in as Will sees data displayed as plain text. Will is not allowed to modify the data, so the form doesn't give him a mechanism to do so. User Ellen does have modify permissions, so when Ellen views the form page, she sees a series of edit boxes which allow her to change field contents. Using access rights in this manner can save you from creating extra pages.

The appearance of some page elements, such as *TAdapterDisplayField*, is determined by its *ViewMode* property. If *ViewMode* is set to *vmToggleOnAccess*, the page element will appear as an edit box to users with modify access. Users without modify access will see plain text. Set the *ViewMode* property to *vmToggleOnAccess* to allow the page element's appearance and function to be determined dynamically.

A Web user list is a list of *TWebUserListItem* objects, one for each user who can login to the system. Permissions for users are stored in their Web user list item's *AccessRights* property. *AccessRights* is a text string, so you are free to specify permissions any way you like. Create a name for every kind of access right you want in your server application. If you want a user to have multiple access rights, separate items in the list with a space, semicolon or comma.

Access rights for fields are controlled by their *ViewAccess* and *ModifyAccess* properties. *ViewAccess* stores the name of the access rights needed to view a given field. *ModifyAccess* dictates what access rights are needed to modify field data. These properties appear in two places: in each field and in the adapter object that contains them.

Checking access rights is a two-step process. When deciding the appearance of a field in a page, the application first checks the field's own access rights. If the value is an empty string, the application then checks the access rights for the adapter which contains the field. If the adapter property is empty as well, the application will follow its default behavior. For modify access, the default behavior is to allow modifications by any user in the Web user list who has a non-empty *AccessRights* property. For view access, permission is automatically granted when no view access rights are specified.

Hiding fields and their contents

You can hide the contents of a field from users who don't have appropriate view permissions. First set the *ViewAccess* property for the field to match the permission you want users to have. Next, make sure that the *ViewAccess* for the field's page element is set to *vmToggleOnAccess*. The field caption will appear, but the value of the field won't.

Of course, it is often best to hide all references to the field when a user doesn't have view permissions. To do so, set the *HideOptions* for the field's page element to include *hoHideOnNoDisplayAccess*. Neither the caption nor the contents of the field will be displayed.

Preventing page access

You may decide that certain pages should not be accessible to unauthorized users. To grant check access rights before displaying pages, alter your call to the *TWebPageInfo* constructor in the Web request handler's AddWebModuleFactory command. This command appears in the initialization section of the source code for your module.

The constructor for TWebPageInfo takes up to 6 arguments. WebSnap usually leaves four of them set to default values (empty strings), so the call generally looks like this:

```
TWebPageInfo.Create([wpPublished, wpLoginRequired], '.html')
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished /* << wpLoginRequired */, ".html", "", "", "", "");
```

To check permissions before granting access, you need to supply the string for the necessary permission in the sixth parameter. For example, let's say that the permission is called "Access". This is how you could modify the creator:

```
TWebPageInfo.Create([wpPublished, wpLoginRequired], '.html', '', '', '', 'Access')
static TWebPageInit WebInit(__classid(TAdapterPageProducerPage3), crOnDemand, caCache,
PageAccess << wpPublished /* << wpLoginRequired */, ".html", "", "", "", "Access");
```

Access to the page will now be denied to anyone who lacks Access permission.

3.2.5.5.5.6 Using the Sessions Service

The sessions service, which is an object of type TSessionsService, keeps track of the users who are logged into your Web server application. The sessions service is responsible for assigning a different session for each user and for associating name/value pairs (such as a username) with a user.

Information contained in a sessions service is stored in the application's memory. Therefore, the Web server application must keep running between requests for the sessions service to work. Some server application types, such as CGI, terminate between requests.

Note: If you want your application to support logins, be sure to use a server type that does not terminate between requests. If your project produces a Web App debugger executable, you must have the application running in the background before it receives a page request. Otherwise it will terminate after each page request, and users will never be able to get past the login page.

There are two important properties in the sessions service which you can use to change default server behavior. The *MaxSessions* property specifies how many users can be logged into the system at any given time. The default value for *MaxSessions* is -1, which places no software limitation on the number of allowed users. Of course, your server hardware can still run short of memory or processor cycles for new users, which can adversely affect system performance. If you are concerned that excessive numbers of users might overwhelm your server, be sure to set *MaxSessions* to an appropriate value.

The *DefaultTimeout* property specifies the defaut time-out period in minutes. After *DefaultTimeout* minutes have passed without any user activity, the session is automatically terminated. If the user had logged in, all login information is lost.. The default value is 20. You can override the default value in any given session by changing its *TimeoutMinutes* property.

3.2.5.5.6 Server-side scripting in WebSnap

Topics

Name	Description
Script Objects (see page 2333)	Script objects are objects that script commands can reference. You make objects available for scripting by registering an *IDispatch* interface to the object with the active scripting engine. The following objects are available for scripting: ***Script objects***

Server-side Scripting in WebSnap (see page 2333)	Page producer templates can include scripting languages such as JScript or VBScript. The page producer executes the script in response to a request for the producer's content. Because the Web server application evaluates the script, it is called server-side script, as opposed to client-side script (which is evaluated by the browser). This topic provides a conceptual overview of server-side scripting and how it is used by WebSnap applications. Although server-side scripting is a valuable part of WebSnap, it is not essential that you use scripting in your WebSnap applications. Scripting is used for HTML generation and nothing else. It allows... more (see page 2333)

3.2.5.5.6.1 Script Objects

Script objects are objects that script commands can reference. You make objects available for scripting by registering an *IDispatch* interface to the object with the active scripting engine. The following objects are available for scripting:

Script objects

Script object	Description
Application	Provides access to the application adapter of the Web Application module.
EndUser	Provides access to the end user adapter of the Web Application module.
Session	Provides access to the session object of the Web Application module.
Pages	Provides access to the application pages.
Modules	Provides access to the application modules.
Page	Provides access to the current page
Producer	Provides access to the page producer of the Web Page module.
Response	Provides access to the WebResponse. Use this object when tag replacement is not desired.
Request	Provides access to the WebRequest.
Adapter objects	All of the adapter components on the current page can be referenced without qualification. Adapters in other modules must be qualified using the Modules objects.

Script objects on the current page, which all use the same adapter, can be referenced without qualification. Script objects on other pages are part of another page module and have a different adapter object. They can be accessed by starting the script object reference with the name of the adapter object. For example,

```
<%= FirstName %>
```

displays the contents of the *FirstName* property of the current page's adapter. The following script line displays the *FirstName* property of Adapter1, which is in another page module:

```
<%= Adapter1.FirstName %>
```

See Also

Type of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.5.6.2 Server-side Scripting in WebSnap

Page producer templates can include scripting languages such as JScript or VBScript. The page producer executes the script in response to a request for the producer's content. Because the Web server application evaluates the script, it is called server-side script, as opposed to client-side script (which is evaluated by the browser).

This topic provides a conceptual overview of server-side scripting and how it is used by WebSnap applications. Although server-side scripting is a valuable part of WebSnap, it is not essential that you use scripting in your WebSnap applications.

Scripting is used for HTML generation and nothing else. It allows you to insert application data into an HTML page. In fact, almost all of the properties exposed by adapters and other script-enabled objects are read-only. Server-side script isn't used to change application data, which is still managed by components and event handlers written in your application's source code.

There are other ways to insert application data into an HTML page. You can use Web Broker's transparent tags or some other tag-based solution, if you prefer. For example, several projects in the WebSnap examples directory use XML and XSL instead of scripting. Without scripting, however, you will be forced to write most of your HTML generation logic in source code, which will increase your development time.

The scripting used in WebSnap is object-oriented and supports conditional logic and looping, which can greatly simplify your page generation tasks. For example, your pages may include a data field that can be edited by some users but not others. With scripting, conditional logic can be placed in your template pages which displays an edit box for authorized users and simple text for others. With a tag-based approach, you must program such decision-making into your HTML generating source code.

Active scripting

WebSnap relies on *active scripting* to implement server-side script. Active scripting is a technology created by Microsoft to allow a scripting language to be used with application objects through COM interfaces. Microsoft ships two active scripting languages, VBScript and JScript. Support for other languages is available through third parties.

Script engine

The page producer's *ScriptEngine* property identifies the active scripting engine that evaluates the script within a template. It is set to support JScript by default, but it can also support other scripting languages (such as VBScript).

Note: WebSnap's adapters are designed to produce JScript. You will need to provide your own script generation logic for other scripting languages.

Script blocks

Script blocks, which appear in HTML templates, are delimited by <% and %>. The script engine evaluates any text inside script blocks. The result becomes part of the page producer's content. The page producer writes text outside of a script block after translating any embedded transparent tags. Script blocks can also enclose text, allowing conditional logic and loops to dictate the output of text. For example, the following JScript block generates a list of five numbered lines:

```
<ul>
<% for (i=0;i<5;i++) { %>
   <li>Item <%=i %></li>
<% } %>
</ul>
```

(The <%= delimiter is short for *Response.Write*.)

Creating script

Developers can take advantage of WebSnap features to automatically generate script.

Wizard templates

When creating a new WebSnap application or page module, WebSnap wizards provide a template field that is used to select the initial content for the page module template. For example, the Default template generates JScript which, in turn, displays the application title, page name, and links to published pages.

TAdapterPageProducer

The TAdapterPageProducer builds forms and tables by generating HTML and JScript. The generated JScript calls adapter objects to retrieve field values, field image parameters, and action parameters.

Editing and viewing script

When the Web Page module uses *TAdapterPageProducer* the page module views become available when this component is double-clicked. You can access the page module view with the HTML resulting from the executed script using the HTML Script tab. The HTML Script tab displays the HTML and JScript generated by the *TAdapterPageProducer* object. Consult this view to

see how to write script that builds HTML forms to display adapter fields and execute adapter actions.

Including script in a page

A template can include script from a file or from another page. To include script from a file, use the following code statement:

```
<!-- #include file="filename.html" -->
```

When the template includes script from another page, the script is evaluated by the including page. Use the following code statement to include the unevaluated content of page1.

```
<!-- #include page="page1" -- >
```

See Also

Types of Web Server Applications (see page 2249)

The Structure of a Web Broker Application (see page 2286)

3.2.5.6 Working with sockets

Topics

Name	Description
Describing sockets (see page 2335)	
Working with Sockets (see page 2337)	The socket components let you create an application that can communicate with other systems using TCP/IP and related protocols. Using sockets, you can read and write over connections to other machines without worrying about the details of the underlying networking software. Sockets provide connections based on the TCP/IP protocol, but are sufficiently general to work with related protocols such as User Datagram Protocol (UDP), Xerox Network System (XNS), Digital's DECnet, or Novell's IPX/SPX family. Using sockets, you can write network servers or client applications that read from and write to other systems. A server or client application is... more (see page 2337)
Implementing services (see page 2338)	
Reading and writing over socket connections (see page 2339)	
Responding to socket events (see page 2341)	
Types of socket connections (see page 2344)	
Using socket components (see page 2346)	

3.2.5.6.1 Describing sockets

Topics

Name	Description
Describing Sockets (see page 2336)	Sockets let your network application communicate with other systems over the network. Each socket can be viewed as an endpoint in a network connection. It has an address that specifies: • The system on which it is running. • The types of interfaces it understands. • The port it is using for the connection. A full description of a socket connection includes the addresses of the sockets on both ends of the connection. You can describe the address of each socket endpoint by supplying both the IP address or host (see page 2336) and the port number (see page 2337). Before you can make a socket connection,... more (see page 2336)

Describing the Host (see page 2336)	The host is the system that is running the application that contains the socket. You can describe the host for a socket by giving its IP address, which is a string of four numeric (byte) values in the standard Internet dot notation, such as
Using Ports (see page 2337)	While the IP address provides enough information to find the system on the other end of a socket connection, you also need a port number on that system. Without port numbers, a system could only form a single connection at a time. Port numbers are unique identifiers that enable a single system to host multiple connections simultaneously, by giving each connection a separate port number. One way to look at port numbers is as numeric codes for the services implemented by network applications. This is a convention that allows listening server connections to make themselves available on a fixed port... more (see page 2337)

3.2.5.6.1.1 Describing Sockets

Sockets let your network application communicate with other systems over the network. Each socket can be viewed as an endpoint in a network connection. It has an address that specifies:

- The system on which it is running.
- The types of interfaces it understands.
- The port it is using for the connection.

A full description of a socket connection includes the addresses of the sockets on both ends of the connection. You can describe the address of each socket endpoint by supplying both the IP address or host (see page 2336) and the port number (see page 2337).

Before you can make a socket connection, you must fully describe the sockets that form its endpoints. Some of the information is available from the system your application is running on. For instance, you do not need to describe the local IP address of a client socket—this information is available from the operating system.

The information you must provide depends on the type of socket you are working with. Client sockets must describe the server they want to connect to. Listening server sockets must describe the port that represents the service they provide.

See Also

Implementing Services (see page 2338)

Types of Socket Connections (see page 2345)

Using Socket Components (see page 2348)

Responding to Socket Events (see page 2343)

Reading and Writing Over Socket Connections (see page 2341)

3.2.5.6.1.2 Describing the Host

The host is the system that is running the application that contains the socket. You can describe the host for a socket by giving its IP address, which is a string of four numeric (byte) values in the standard Internet dot notation, such as

```
123.197.1.2
```

A single system may support more than one IP address.

IP addresses are often difficult to remember and easy to mistype. An alternative is to use the host name. Host names are aliases for the IP address that you often see in Uniform Resource Locators (URLs). They are strings containing a domain name and service, such as

```
http://www.ASite.com
```

Most Intranets provide host names for the IP addresses of systems on the Internet. You can learn the host name associated with any IP address (if one already exists) by executing the following command from a command prompt:

```
nslookup IPADDRESS
```

where IPADDRESS is the IP address you're interested in. If your local IP address doesn't have a host name and you decide you want one, contact your network administrator. It is common for computers to refer to themselves with the name *localhost* and the IP number 127.0.0.1.

Server sockets do not need to specify a host. The local IP address can be read from the system. If the local system supports more than one IP address, server sockets will listen for client requests on all IP addresses simultaneously. When a server socket accepts a connection, the client socket provides the remote IP address.

Client sockets must specify the remote host by providing either its host name or IP address.

Choosing between a host name and an IP address

Most applications use the host name to specify a system. Host names are easier to remember, and easier to check for typographical errors. Further, servers can change the system or IP address that is associated with a particular host name. Using a host name allows the client socket to find the abstract site represented by the host name, even when it has moved to a new IP address.

If the host name is unknown, the client socket must specify the server system using its IP address. Specifying the server system by giving the IP address is faster. When you provide the host name, the socket must search for the IP address associated with the host name, before it can locate the server system.

See Also

Using Ports (see page 2337)

Parts of a Uniform Resource Locator (see page 2253)

3.2.5.6.1.3 Using Ports

While the IP address provides enough information to find the system on the other end of a socket connection, you also need a port number on that system. Without port numbers, a system could only form a single connection at a time. Port numbers are unique identifiers that enable a single system to host multiple connections simultaneously, by giving each connection a separate port number.

One way to look at port numbers is as numeric codes for the services implemented by network applications. This is a convention that allows listening server connections to make themselves available on a fixed port number so that they can be found by client sockets. Server sockets listen on the port number associated with the service they provide. When they accept a connection to a client socket, they create a separate socket connection that uses a different, arbitrary, port number. This way, the listening connection can continue to listen on the port number associated with the service.

Client sockets use an arbitrary local port number, as there is no need for them to be found by other sockets. They specify the port number of the server socket to which they want to connect so that they can find the server application. Often, this port number is specified indirectly, by naming the desired service.

See Also

Describing the Host (see page 2336)

3.2.5.6.2 Working with Sockets

The socket components let you create an application that can communicate with other systems using TCP/IP and related protocols. Using sockets, you can read and write over connections to other machines without worrying about the details of the underlying networking software. Sockets provide connections based on the TCP/IP protocol, but are sufficiently general to work with related protocols such as User Datagram Protocol (UDP), Xerox Network System (XNS), Digital's DECnet, or Novell's IPX/SPX family.

Using sockets, you can write network servers or client applications that read from and write to other systems. A server or client application is usually dedicated to a single service such as Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP).

Using server sockets, an application that provides one of these services can link to client applications that want to use that service. Client sockets allow an application that uses one of these services to link to server applications that provide the service.

To write applications that use sockets, you should understand

- Implementing services (see page 2338)
- Types of socket connections (see page 2345)
- Describing sockets (see page 2336)
- Using socket components
- Responding to socket events (see page 2343)
- Reading and writing over socket connections (see page 2341)

See Also

Creating Internet Server Applications (see page 2251)

3.2.5.6.3 Implementing services

Topics

Name	Description
Implementing Services (see page 2338)	Sockets provide one of the pieces you need to write network servers or client applications. For many services, such as HTTP or FTP, third party servers are readily available. Some are even bundled with the operating system, so that there is no need to write one yourself. However, when you want more control over the way the service is implemented, a tighter integration between your application and the network communication, or when no server is available for the particular service you need, then you may want to create your own server or client application. For example, when working with... more (see page 2338)
Services and Ports (see page 2339)	Most standard services are associated, by convention, with specific port numbers. When implementing services, you can consider the port number a numeric code for the service.
Understanding Service Protocols (see page 2339)	Before you can write a network server or client, you must understand the service that your application is providing or using. Many services have standard protocols that your network application must support. If you are writing a network application for a standard service such as HTTP, FTP, or even finger or time, you must first understand the protocols used to communicate with other systems. See the documentation on the particular service you are providing or using. If you are providing a new service for an application that communicates with other systems, the first step is designing the communication protocol for... more (see page 2339)

3.2.5.6.3.1 Implementing Services

Sockets provide one of the pieces you need to write network servers or client applications. For many services, such as HTTP or FTP, third party servers are readily available. Some are even bundled with the operating system, so that there is no need to write one yourself. However, when you want more control over the way the service is implemented, a tighter integration between your application and the network communication, or when no server is available for the particular service you need, then you may want to create your own server or client application. For example, when working with distributed data sets, you may want to write a layer to communicate with databases on other systems.

To implement or use a service using sockets, you must understand

- service protocols (see page 2339)
- services and ports (see page 2339)

See Also

Types of Socket Connections (see page 2345)

Describing Sockets (see page 2336)

Using Socket Components (see page 2348)

Responding to Socket Events (see page 2343)

Reading and Writing Over Socket Connections (see page 2341)

3.2.5.6.3.2 Services and Ports

Most standard services are associated, by convention, with specific port numbers. When implementing services, you can consider the port number a numeric code for the service.

See Also

Service Protocols (see page 2339)

Using Ports (see page 2337)

3.2.5.6.3.3 Understanding Service Protocols

Before you can write a network server or client, you must understand the service that your application is providing or using. Many services have standard protocols that your network application must support. If you are writing a network application for a standard service such as HTTP, FTP, or even finger or time, you must first understand the protocols used to communicate with other systems. See the documentation on the particular service you are providing or using.

If you are providing a new service for an application that communicates with other systems, the first step is designing the communication protocol for the servers and clients of this service. What messages are sent? How are these messages coordinated? How is the information encoded?

Communicating with applications

Often, your network server or client application provides a layer between the networking software and an application that uses the service. For example, an HTTP server sits between the Internet and a Web server application that provides content and responds to HTTP request messages.

Sockets provide the interface between your network server or client application and the networking software. You must provide the interface between your application and the clients that use it. You can copy the API of a standard third party server (such as Apache), or you can design and publish your own API.

See Also

Services and Ports (see page 2339)

HTTP Server Activity (see page 2248)

3.2.5.6.4 Reading and writing over socket connections

Topics

Name	Description
Blocking Connections (see page 2340)	When the connection is blocking, your socket must initiate reading or writing over the connection. It cannot wait passively for a notification from the socket connection. Use a blocking socket when your end of the connection is in charge of when reading and writing takes place. For client or server sockets, set the BlockMode property to *bmBlocking* to form a blocking connection. Depending on what else your client application does, you may want to create a new execution thread for reading or writing, so that your application can continue executing code on other threads while it waits for the reading... more (see page 2340)

Non-blocking Connections (see page 2340)	Non-blocking connections read and write asynchronously, so that the transfer of data does not block the execution of other code in you network application. To create a non-blocking connection for client or server sockets, set the BlockMode property to *bmNonBlocking.* When the connection is non-blocking, reading and writing events (see page 2340) inform your socket when the socket on the other end of the connection tries to read or write information.
Reading and Writing Events (see page 2340)	Non-blocking sockets generate reading and writing events when they need to read or write over the connection. You can respond to these notifications in an *OnReceive* or *OnSend* event handler. The socket object associated with the socket connection is provided as a parameter to the read or write event handlers. This socket object provides a number of methods to allow you to read or write over the connection. To read from the socket connection, use the ReceiveBuf or ReceiveIn method. To write to the socket connection, use the SendBuf, SendStream, or SendIn.
Reading and Writing Over Socket Connections (see page 2341)	The reason you form socket connections to other machines is so that you can read or write information over those connections. What information you read or write, or when you read it or write it, depends on the service associated with the socket connection. Reading and writing over sockets can occur asynchronously, so that it does not block the execution of other code in your network application. This is called a non-blocking connection (see page 2340). You can also form blocking connections (see page 2340), where your application waits for the reading or writing to be completed before executing the next line of code.... more (see page 2341)

3.2.5.6.4.1 Blocking Connections

When the connection is blocking, your socket must initiate reading or writing over the connection. It cannot wait passively for a notification from the socket connection. Use a blocking socket when your end of the connection is in charge of when reading and writing takes place.

For client or server sockets, set the BlockMode property to *bmBlocking* to form a blocking connection. Depending on what else your client application does, you may want to create a new execution thread for reading or writing, so that your application can continue executing code on other threads while it waits for the reading or writing over the connection to be completed.

For server sockets, set the *BlockMode* property to *bmBlocking* or *bmThreadBlocking* to form a blocking connection. Because blocking connections hold up the execution of all other code while the socket waits for information to be written or read over the connection, server socket components always spawn a new execution thread for every client connection when the *BlockMode* is *bmThreadBlocking*. When the *BlockMode* is *bmBlocking*, program execution is blocked until a new connection is established.

See Also

Non-blocking Connections (see page 2340)

3.2.5.6.4.2 Non-blocking Connections

Non-blocking connections read and write asynchronously, so that the transfer of data does not block the execution of other code in you network application. To create a non-blocking connection for client or server sockets, set the BlockMode property to *bmNonBlocking.*

When the connection is non-blocking, reading and writing events (see page 2340) inform your socket when the socket on the other end of the connection tries to read or write information.

See Also

Blocking Connections (see page 2340)

Responding to Socket Events (see page 2343)

3.2.5.6.4.3 Reading and Writing Events

Non-blocking sockets generate reading and writing events when they need to read or write over the connection. You can

respond to these notifications in an *OnReceive* or *OnSend* event handler.

The socket object associated with the socket connection is provided as a parameter to the read or write event handlers. This socket object provides a number of methods to allow you to read or write over the connection.

To read from the socket connection, use the ReceiveBuf or ReceiveIn method. To write to the socket connection, use the SendBuf, SendStream, or SendIn.

See Also

Error Events (see page 2342)

Client Events (see page 2342)

Server Events (see page 2343)

Types of Socket Connections (see page 2345)

3.2.5.6.4.4 Reading and Writing Over Socket Connections

The reason you form socket connections to other machines is so that you can read or write information over those connections. What information you read or write, or when you read it or write it, depends on the service associated with the socket connection.

Reading and writing over sockets can occur asynchronously, so that it does not block the execution of other code in your network application. This is called a non-blocking connection (see page 2340). You can also form blocking connections (see page 2340), where your application waits for the reading or writing to be completed before executing the next line of code.

See Also

Implementing Services (see page 2338)

Types of Socket Connections (see page 2345)

Describing Sockets (see page 2336)

Using Socket Components (see page 2348)

Responding to Socket Events (see page 2343)

3.2.5.6.5 Responding to socket events

Topics

Name	Description
Client Events (see page 2342)	When a client socket opens a connection, the following events occur: • The socket is set up and initialized for event notification. • An OnCreateHandle event occurs after the server and server socket is created. At this point, the socket object available through the *Handle* property can provide information about the server or client socket that will form the other end of the connection. This is the first chance to obtain the actual port used for the connection, which may differ from the port of the listening sockets that accepted the connection. • The connection request is accepted by the server and completed... more (see page 2342)

Error Events (see page 2342)	Client and server sockets generate OnError events when they receive error messages from the connection. You can write an *OnError* event handler to respond to these error messages. The event handler is passed information about • What socket object received the error notification. • What the socket was trying to do when the error occurred. • The error code that was provided by the error message. You can respond to the error in the event handler, and change the error code to 0 to prevent the socket from raising an exception.
Responding to Socket Events (see page 2343)	When writing applications that use sockets, you can write or read to the socket anywhere in the program. You can write to the socket using the *SendBuf*, *SendStream*, or *Sendln* methods in your program after the socket has been opened. You can read from the socket using the similarly-named methods *ReceiveBuf* and *Receiveln*. The *OnSend* and *OnReceive* events are triggered every time something is written or read from the socket. They can be used for filtering. Every time you read or write, a read or write event is triggered. Both client sockets and server sockets generate error (see page 2342)... more (see page 2343)
Server Events (see page 2343)	Server socket components form two types of connections: listening connections and connections to client applications. The server socket receives events during the formation of each of these connections.

3.2.5.6.5.1 Client Events

When a client socket opens a connection, the following events occur:

- The socket is set up and initialized for event notification.
- An OnCreateHandle event occurs after the server and server socket is created. At this point, the socket object available through the *Handle* property can provide information about the server or client socket that will form the other end of the connection. This is the first chance to obtain the actual port used for the connection, which may differ from the port of the listening sockets that accepted the connection.
- The connection request is accepted by the server and completed by the client socket.
- When the connection is established, the OnConnect notification event occurs.

See Also

Error Events (see page 2342)

Server Events (see page 2343)

Reading and Writing Events (see page 2340)

Using Client Sockets (see page 2348)

Reading and Writing Over Socket Connections (see page 2341)

Using Ports (see page 2337)

3.2.5.6.5.2 Error Events

Client and server sockets generate OnError events when they receive error messages from the connection. You can write an *OnError* event handler to respond to these error messages. The event handler is passed information about

- What socket object received the error notification.
- What the socket was trying to do when the error occurred.
- The error code that was provided by the error message.

You can respond to the error in the event handler, and change the error code to 0 to prevent the socket from raising an exception.

See Also

Client Events (see page 2342)

Server Events (see page 2343)

Reading and Writing Events (see page 2340)

3.2.5.6.5.3 Responding to Socket Events

When writing applications that use sockets, you can write or read to the socket anywhere in the program. You can write to the socket using the *SendBuf*, *SendStream*, or *Sendln* methods in your program after the socket has been opened. You can read from the socket using the similarly-named methods *ReceiveBuf* and *Receiveln*. The *OnSend* and *OnReceive* events are triggered every time something is written or read from the socket. They can be used for filtering. Every time you read or write, a read or write event is triggered.

Both client sockets and server sockets generate error events (see page 2342) when they receive error messages from the connection.

Socket components also receive two events in the course of opening and completing a connection. If your application needs to influence how the opening of the socket proceeds, you must use the *SendBuf* and *ReceiveBuf* methods to respond to these client events (see page 2342) or server events (see page 2343).

See Also

Implementing Services (see page 2338)

Types of Socket Connections (see page 2345)

Describing Sockets (see page 2336)

Using Socket Components (see page 2348)

Reading and Writing Over Socket Connections (see page 2341)

3.2.5.6.5.4 Server Events

Server socket components form two types of connections: listening connections and connections to client applications. The server socket receives events during the formation of each of these connections.

Events when listening

Just before the listening connection is formed, the OnListening event occurs. You can use its Handle property to make changes to the socket before it is opened for listing. For example, if you want to restrict the IP addresses the server uses for listening, you would do that in an *OnListening* event handler.

Events with client connections

When a server socket accepts a client connection request, the following events occur:

- An OnAccept event occurs, passing in the new *TTcpClient* object to the event handler. This is the first point when you can use the properties of *TTcpClient* to obtain information about the server endpoint of the connection to a client.
- If BlockMode is *bmThreadBlocking* an OnGetThread event occurs. If you want to provide your own customized descendant of ServerSocketThread, you can create one in an *OnGetThread* event handler, and that will be used instead of *TServerSocketThread*. If you want to perform any initialization of the thread, or make any socket API calls before the thread starts reading or writing over the connection, you should use the *OnGetThread* event handler for these tasks as well.
- The client completes the connection and an OnAccept event occurs. With a non-blocking server, you may want to start reading or writing over the socket connection at this point.

See Also

Error Events (see page 2342)

Client Events (see page 2342)

Reading and Writing Events (see page 2340)

Using Server Sockets (see page 2351)

Reading and Writing Over Socket Connections (see page 2341)

Types of Socket Connections (see page 2345)

3.2.5.6.6 Types of socket connections

Topics

Name	Description
Client Connections (see page 2344)	Client connections connect a client socket on the local system to a server socket on a remote system. Client connections are initiated by the client socket. First, the client socket must describe the server socket to which it wishes to connect. The client socket then looks up the server socket and, when it locates the server, requests a connection. The server socket may not complete the connection right away. Server sockets maintain a queue of client requests, and complete connections as they find time. When the server socket accepts the client connection, it sends the client socket a full description... more (see page 2344)
Listening Connections (see page 2345)	Server sockets do not locate clients. Instead, they form passive "half connections" that listen for client requests. Server sockets associate a queue with their listening connections; the queue records client connection requests as they come in. When the server socket accepts a client connection request, it forms a new socket to connect to the client, so that the listening connection can remain open to accept other client requests.
Server Connections (see page 2345)	Server connections are formed by server sockets when a listening socket accepts a client request. A description of the server socket that completes the connection to the client is sent to the client when the server accepts the connection. The connection is established when the client socket receives this description and completes the connection.
Types of Socket Connections (see page 2345)	Socket connections can be divided into three basic types, which reflect how the connection was initiated and what the local socket is connected to. These are • Client connections (see page 2344). • Listening connections (see page 2345). • Server connections (see page 2345). Once the connection to a client socket is completed, the server connection is indistinguishable from a client connection. Both end points have the same capabilities and receive the same types of events. Only the listening connection is fundamentally different, as it has only a single endpoint.

3.2.5.6.6.1 Client Connections

Client connections connect a client socket on the local system to a server socket on a remote system. Client connections are initiated by the client socket. First, the client socket must describe the server socket to which it wishes to connect. The client socket then looks up the server socket and, when it locates the server, requests a connection. The server socket may not complete the connection right away. Server sockets maintain a queue of client requests, and complete connections as they find time. When the server socket accepts the client connection, it sends the client socket a full description of the server socket to which it is connecting, and the connection is completed by the client.

See Also

Listening Connections (see page 2345)

Server Connections (see page 2345)

Using Client Sockets (see page 2348)

Client Events (see page 2342)

3.2.5.6.6.2 Listening Connections

Server sockets do not locate clients. Instead, they form passive "half connections" that listen for client requests. Server sockets associate a queue with their listening connections; the queue records client connection requests as they come in. When the server socket accepts a client connection request, it forms a new socket to connect to the client, so that the listening connection can remain open to accept other client requests.

See Also

Client Connections (see page 2344)

Server Connections (see page 2345)

Listening for Client Requests (see page 2350)

Server Events (see page 2343)

3.2.5.6.6.3 Server Connections

Server connections are formed by server sockets when a listening socket accepts a client request. A description of the server socket that completes the connection to the client is sent to the client when the server accepts the connection. The connection is established when the client socket receives this description and completes the connection.

See Also

Client Connections (see page 2344)

Listening Connections (see page 2345)

Connecting to Clients (see page 2350)

Server Events (see page 2343)

3.2.5.6.6.4 Types of Socket Connections

Socket connections can be divided into three basic types, which reflect how the connection was initiated and what the local socket is connected to. These are

- Client connections (see page 2344).
- Listening connections (see page 2345).
- Server connections (see page 2345).

Once the connection to a client socket is completed, the server connection is indistinguishable from a client connection. Both end points have the same capabilities and receive the same types of events. Only the listening connection is fundamentally different, as it has only a single endpoint.

See Also

Implementing Services (see page 2338)

Describing Sockets (see page 2336)

Using Socket Components (see page 2348)

Responding to Socket Events (see page 2343)

Reading and Writing Over Socket Connections (see page 2341)

3.2.5.6.7 Using socket components

Topics

Name	Description
Using client sockets (see page 2346)	
Using Socket Components (see page 2348)	The Internet category includes three socket components that allow your network application to form connections to other machines, and that allow you to read and write information over that connection. These are: • TTcpServer • TTcpClient • TUdpSocket Associated with each of these socket components are socket objects, which represent the endpoint of an actual socket connection. The socket components use the socket objects to encapsulate the socket server calls, so that your application does not need to be concerned with the details of establishing the connection or managing the socket messages. If you want to customize the details of the connections that... more (see page 2348)
Using server sockets (see page 2349)	
Getting Information About the Client or Server Socket Connection (see page 2351)	After completing the connection to a client or server socket, you can use the client or server socket object associated with your socket component to obtain information about the connection. Use the LocalHost and LocalPort properties to determine the address and port number used by the local client or server socket, or use the RemoteHost and RemotePort properties to determine the address and port number used by the remote client or server socket. Use the GetSocketAddr method to build a valid socket address based on the host name and port number. You can use the LookupPort method to look up... more (see page 2351)

3.2.5.6.7.1 Using client sockets

Topics

Name	Description
Closing the Connection (see page 2347)	When you have finished communicating with a server application over the socket connection, you can shut down the connection by calling the Close method. The connection may also be closed from the server end. If that is the case, you will receive notification in an OnDisconnect event.
Forming the Connection (see page 2347)	Once you have set the properties of your client socket component to describe the server you want to connect to, you can form the connection at runtime by calling the Open method. If you want your application to form the connection automatically when it starts up, set the Active property to *True* at design time, using the **Object Inspector**.
Getting Information About the Client Socket Connection (see page 2347)	After completing the connection to a server socket, you can use the client socket object associated with your client socket component to obtain information about the connection. Use the LocalHost and LocalPort properties to determine the address and port number used by the client and server sockets to form the end points of the connection. You can use the Handle property to obtain a handle to the socket connection to use when making socket calls.

Specifying the Desired Server (see page 2348)	Client socket components have a number of properties that allow you to specify the server system and port to which you want to connect. Use the RemoteHost property to specify the remote host server by either its host name or IP address. In addition to the server system, you must specify the port on the server system that your client socket will connect to. You can use the RemotePort property to specify the server port number directly or indirectly by naming the target service.
Using Client Sockets (see page 2348)	Add a TTcpClient or TUdpSocket component to your form or data module to turn your application into a TCP/IP or UDP client. Client sockets allow you to specify the server socket you want to connect to, and the service you want that server to provide. Once you have described the desired connection, you can use the client socket component to complete the connection to the server. Each client socket component uses a single client socket object to represent the client endpoint in a connection. Use client sockets to • Specify the desired server (see page 2348). • Connect to the server (see page 2347). • Get information (see page 2347)... more (see page 2348)

3.2.5.6.7.1.1 Closing the Connection

When you have finished communicating with a server application over the socket connection, you can shut down the connection by calling the Close method. The connection may also be closed from the server end. If that is the case, you will receive notification in an OnDisconnect event.

See Also

Specifying the Desired Server (see page 2348)

Forming the Connection (see page 2347)

Reading and Writing Over Socket Connections (see page 2341)

Getting Information About the Connection (see page 2347)

Client Events (see page 2342)

3.2.5.6.7.1.2 Forming the Connection

Once you have set the properties of your client socket component to describe the server you want to connect to, you can form the connection at runtime by calling the Open method. If you want your application to form the connection automatically when it starts up, set the Active property to *True* at design time, using the **Object Inspector**.

See Also

Specifying the Desired Server (see page 2348)

Getting Information About the Connection (see page 2347)

Reading and Writing Over Socket Connections (see page 2341)

Closing the Connection (see page 2347)

3.2.5.6.7.1.3 Getting Information About the Client Socket Connection

After completing the connection to a server socket, you can use the client socket object associated with your client socket component to obtain information about the connection. Use the LocalHost and LocalPort properties to determine the address and port number used by the client and server sockets to form the end points of the connection. You can use the Handle property to obtain a handle to the socket connection to use when making socket calls.

See Also

Specifying the Desired Server (see page 2348)

Forming the Connection (see page 2347)

Reading and Writing Over Socket Connections (see page 2341)

Closing the Connection (see page 2347)

3.2.5.6.7.1.4 Specifying the Desired Server

Client socket components have a number of properties that allow you to specify the server system and port to which you want to connect. Use the RemoteHost property to specify the remote host server by either its host name or IP address.

In addition to the server system, you must specify the port on the server system that your client socket will connect to. You can use the RemotePort property to specify the server port number directly or indirectly by naming the target service.

See Also

Forming the Connection (see page 2347)

Getting Information About the Connection (see page 2347)

Reading and Writing Over Socket Connections (see page 2341)

Closing the Connection (see page 2347)

Describing Sockets (see page 2336)

3.2.5.6.7.1.5 Using Client Sockets

Add a TTcpClient or TUdpSocket component to your form or data module to turn your application into a TCP/IP or UDP client. Client sockets allow you to specify the server socket you want to connect to, and the service you want that server to provide. Once you have described the desired connection, you can use the client socket component to complete the connection to the server.

Each client socket component uses a single client socket object to represent the client endpoint in a connection.

Use client sockets to

- Specify the desired server (see page 2348).
- Connect to the server (see page 2347).
- Get information about the connection (see page 2347).
- Read from or write to the server (see page 2341).
- Close the connection (see page 2347).

See Also

Using Server Sockets (see page 2351)

Client Events (see page 2342)

3.2.5.6.7.2 Using Socket Components

The Internet category includes three socket components that allow your network application to form connections to other machines, and that allow you to read and write information over that connection. These are:

- TTcpServer
- TTcpClient

- TUdpSocket

Associated with each of these socket components are socket objects, which represent the endpoint of an actual socket connection. The socket components use the socket objects to encapsulate the socket server calls, so that your application does not need to be concerned with the details of establishing the connection or managing the socket messages.

If you want to customize the details of the connections that the socket components make on your behalf, you can use the properties, events, and methods of the socket objects.

See Also

Specifying the Desired Server (see page 2348)

Forming the Connection (see page 2347)

Reading and Writing Over Socket Connections (see page 2341)

Closing the Connection (see page 2347)

Reading and Writing Over Socket Connections (see page 2341)

3.2.5.6.7.3 Using server sockets

Topics

Name	Description
Closing Server Connections (see page 2349)	When you want to shut down the listening connection, call the Close method or set the Active property to *False*. This shuts down all open connections to client applications, cancels any pending connections that have not been accepted, and then shuts down the listening connection so that your server socket component does not accept any new connections. When TCP clients shut down their individual connections to your server socket, you are informed by an OnDisconnect event.
Connecting to Clients (see page 2350)	A listening server socket component automatically accepts client connection requests when they are received. You receive notification every time this occurs in an OnAccept event.
Listening for Client Requests (see page 2350)	Once you have set the port number of your server socket component, you can form a listening connection at runtime by calling the Open method. If you want your application to form the listening connection automatically when it starts up, set the Active property to *True* at design time, using the **Object Inspector**.
Specifying the Port (see page 2350)	Before your server socket can listen to client requests, you must specify the port that your server will listen on. You can specify this port using the LocalPort property. If your server application is providing a standard service that is associated by convention with a specific port number, you can also specify the service name using the *LocalPort* property. It is a good idea to use the service name instead of a port number, because it is easy to introduce typographical errors when specifying the port number.
Using Server Sockets (see page 2351)	Add a server socket component (TTcpServer or TUdpSocket) to your form or data module to turn your application into an IP server. Server sockets allow you to specify the service you are providing or the port you want to use to listen for client requests. You can use the server socket component to listen for and accept client connection requests. Each server socket component uses a single server socket object to represent the server endpoint in a listening connection. It also uses a server client socket object for the server endpoint of each active connection to a client socket that... more (see page 2351)

3.2.5.6.7.3.1 Closing Server Connections

When you want to shut down the listening connection, call the Close method or set the Active property to *False*. This shuts down all open connections to client applications, cancels any pending connections that have not been accepted, and then shuts down the listening connection so that your server socket component does not accept any new connections.

When TCP clients shut down their individual connections to your server socket, you are informed by an OnDisconnect event.

See Also

Specifying the Port (see page 2350)

Listening for Client Requests (see page 2350)

Connecting to Clients (see page 2350)

Reading and Writing Over Socket Connections (see page 2341)

Server Events (see page 2343)

3.2.5.6.7.3.2 Connecting to Clients

A listening server socket component automatically accepts client connection requests when they are received. You receive notification every time this occurs in an OnAccept event.

See Also

Specifying the Port (see page 2350)

Listening for Client Requests (see page 2350)

Reading and Writing Over Socket Connections (see page 2341)

Closing Server Connections (see page 2349)

Server Events (see page 2343)

Server Connections (see page 2345)

3.2.5.6.7.3.3 Listening for Client Requests

Once you have set the port number of your server socket component, you can form a listening connection at runtime by calling the Open method. If you want your application to form the listening connection automatically when it starts up, set the Active property to *True* at design time, using the **Object Inspector**.

See Also

Specifying the Port (see page 2350)

Connecting to Clients (see page 2350)

Reading and Writing Over Socket Connections (see page 2341)

Closing Server Connections (see page 2349)

Listening Connections (see page 2345)

3.2.5.6.7.3.4 Specifying the Port

Before your server socket can listen to client requests, you must specify the port that your server will listen on. You can specify this port using the LocalPort property. If your server application is providing a standard service that is associated by convention with a specific port number, you can also specify the service name using the *LocalPort* property. It is a good idea to use the service name instead of a port number, because it is easy to introduce typographical errors when specifying the port number.

See Also

Listening for Client Requests (see page 2350)

Connecting to Clients (see page 2350)

Reading and Writing Over Socket Connections (see page 2341)

Closing Server Connections (see page 2349)

Services and Ports (see page 2339)

3.2.5.6.7.3.5 Using Server Sockets

Add a server socket component (TTcpServer or TUdpSocket) to your form or data module to turn your application into an IP server. Server sockets allow you to specify the service you are providing or the port you want to use to listen for client requests. You can use the server socket component to listen for and accept client connection requests.

Each server socket component uses a single server socket object to represent the server endpoint in a listening connection. It also uses a server client socket object for the server endpoint of each active connection to a client socket that the server accepts.

Use server sockets to

- Specify the port (see page 2350).
- Listen for client requests (see page 2350).
- Connect to clients (see page 2350).
- Read from or write to the server (see page 2341).
- Close server connections (see page 2349).

See Also

Using Client Sockets (see page 2348)

Server Events (see page 2343)

3.2.5.6.7.4 Getting Information About the Client or Server Socket Connection

After completing the connection to a client or server socket, you can use the client or server socket object associated with your socket component to obtain information about the connection. Use the LocalHost and LocalPort properties to determine the address and port number used by the local client or server socket, or use the RemoteHost and RemotePort properties to determine the address and port number used by the remote client or server socket. Use the GetSocketAddr method to build a valid socket address based on the host name and port number. You can use the LookupPort method to look up the port number. Use the LookupProtocol method to look up the protocol number. Use the LookupHostName method to look up the host name based on the host machine's IP address.

To view network traffic in and out of the socket, use the BytesSent and BytesReceived properties.

See Also

Specifying the Desired Server (see page 2348)

Forming the Connection (see page 2347)

Reading and Writing Over Socket Connections (see page 2341)

Closing the Connection (see page 2347)

3.2.5.7 Working with XML documents

Topics

Name	Description
Working with XML Documents (see page 2352)	XML (Extensible Markup Language) is a markup language for describing structured data. It is similar to HTML, except that the tags describe the structure of information rather than its display characteristics. XML documents provide a simple, text-based way to store information so that it is easily searched or edited. They are often used as a standard, transportable format for data in Web applications, business-to-business communication, and so on. XML documents provide a hierarchical view of a body of data. Tags in the XML document describe the role or meaning of each data element, as illustrated in the following document, which... more (see page 2352)

Using the Document Object Model (see page 2353)	The Document Object Model (DOM) is a set of standard interfaces for representing a parsed XML document. These interfaces are implemented by a number of different third-party vendors. If you do not want to use the default vendor that ships with Delphi, there is a registration mechanism that lets you integrate additional DOM implementations by other vendors into the XML framework. The XMLDOM unit includes declarations for all the DOM interfaces defined in the W3C XML DOM level 2 specification. Each DOM vendor provides an implementation for these interfaces. • To use one of the DOM vendors for which Delphi already... more (see page 2353)
Working with XML Components (see page 2354)	The VCL defines a number of classes and interfaces for working with XML documents. These simplify the process of loading, editing, and saving XML documents. To use the XML classes for examining or editing an XML document you start by setting up an instance of TXMLDocument (see page 2354). You can then work with the nodes (see page 2355) of the XML document component to examine or edit the body of the XML document.
Using TXMLDocument (see page 2354)	The starting point for working with an XML document is the TXMLDocument component.
Working with XML Nodes (see page 2355)	Once an XML document has been parsed by a DOM implementation, the data it represents is available as a hierarchy of nodes. Each node corresponds to a tagged element in the document. For example, given the following XML:
Abstracting XML Documents with the Data Binding Wizard (see page 2356)	It is possible to work with an XML document using only the *TXMLDocument* component and the *IXMLNode* interface it surfaces for the nodes in that document, or even to work exclusively with the DOM interfaces (avoiding even *TXMLDocument*). However, you can write code that is much simpler and more readable by using the XML Data Binding wizard. The Data Binding wizard takes an XML schema or data file and generates a set of interfaces that map on top of it. For example, given XML data that looks like the following:
Using the XML Data Binding Wizard (see page 2358)	
Using Code That the XML Data Binding Wizard Generates (see page 2359)	Once the wizard has generated a set of interfaces and implementation classes, you can use them to work with XML documents that match the structure of the document or schema you supplied to the wizard. Just as when you are using only the built-in XML components (see page 2354), your starting point is the TXMLDocument component that appears on the Internet category of the **Tool Palette**.

3.2.5.7.1 Working with XML Documents

XML (Extensible Markup Language) is a markup language for describing structured data. It is similar to HTML, except that the tags describe the structure of information rather than its display characteristics. XML documents provide a simple, text-based way to store information so that it is easily searched or edited. They are often used as a standard, transportable format for data in Web applications, business-to-business communication, and so on.

XML documents provide a hierarchical view of a body of data. Tags in the XML document describe the role or meaning of each data element, as illustrated in the following document, which describes a collection of stock holdings:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

This example illustrates a number of typical elements in an XML document. The first line is a processing instruction called an XML declaration. The XML declaration is optional but you should include it, because it supplies useful information about the document. In this example, the XML declaration says that the document conforms to version 1.0 of the XML specification, that it uses UTF-8 character encoding, and that it relies on an external file for its document type declaration (DTD).

The second line, which begins with the <!DOCType> tag, is a document type declaration (DTD). The DTD is how XML defines the structure of the document. It imposes syntax rules on the elements (tags) contained in the document. The DTD in this example references another file (sth.dtd). In this case, the structure is defined in an external file, rather than in the XML document itself. Other types of files that describe the structure of an XML document include Reduced XML Data (XDR) and XML schemas (XSD).

The remaining lines are organized into a hierarchy with a single root node (the <StockHoldings> tag). Each node in this hierarchy contains either a set of child nodes, or a text value. Some of the tags (the <Stock> and <shares> tags) include attributes, which are Name=Value pairs that provide details on how to interpret the tag.

Although it is possible to work directly with the text in an XML document, typically applications use additional tools for parsing and editing the data. W3C defines a set of standard interfaces for representing a parsed XML document called the Document Object Model (DOM) (see page 2353). A number of vendors provide XML parsers that implement the DOM interfaces to let you interpret and edit XML documents more easily.

Delphi provides a number of additional tools for working with XML documents. These tools use a DOM parser that is provided by another vendor, and make it even easier to work with XML documents.They include

- VCL components and interfaces (see page 2354) for working with XML documents.
- An XML Data Binding wizard (see page 2356) for generating classes to represent a particular XML document.
- Tools and components for converting between XML documents and data packets (see page 1847), which let you integrate XML documents into database applications.

See Also

Using XML in Database Applications (see page 1847)

3.2.5.7.2 Using the Document Object Model

The Document Object Model (DOM) is a set of standard interfaces for representing a parsed XML document. These interfaces are implemented by a number of different third-party vendors. If you do not want to use the default vendor that ships with Delphi, there is a registration mechanism that lets you integrate additional DOM implementations by other vendors into the XML framework.

The XMLDOM unit includes declarations for all the DOM interfaces defined in the W3C XML DOM level 2 specification. Each DOM vendor provides an implementation for these interfaces.

- To use one of the DOM vendors for which Delphi already includes support, locate the unit that represents the DOM implementation. These units end in the string 'xmldom.' For example, the unit for the Microsoft implementation is MSXMLDOM, the unit for the Xerces implementation is XERCESXMLDOM, and the unit for the Open XML implementation is OXMLDOM. If you add the unit for the desired implementation to your project, the DOM implementation is automatically registered so that it is available to your code.
- To use another DOM implementation, you must create a unit that defines a descendant of the TDOMVendor class. This unit can then work like one of the built-in DOM implementations, making your DOM implementation available when it is included in a project.
- In your descendant class, you must override two methods: the *Description* method, which returns a string identifying the vendor, and the *DOMImplementation* method, which returns the top-level interface (*IDOMImplementation*).
- Your new unit must register the vendor by calling the global RegisterDOMVendor procedure. This call typically goes in the initialization section of the unit.
- When your unit is unloaded, it needs to unregister itself to indicate that the DOM implementation is no longer available.

Unregister the vendor by calling the global UnRegisterDOMVendor procedure. This call typically goes in the finalization section.

Some vendors supply extensions to the standard DOM interfaces. To allow you to uses these extensions, the XMLDOM unit also defines an *IDOMNodeEx* interface. *IDOMNodeEx* is a descendant of the standard *IDOMNode* that includes the most useful of these extensions.

You can work directly with the DOM interfaces to parse and edit XML documents. Simply call the GetDOM function to obtain an *IDOMImplementation* interface, which you can use as a starting point.

Note: For detailed descriptions of the DOM interfaces, see the declarations in the XMLDOM unit, the documentation supplied by your DOM Vendor, or the specifications provided on the W3C web site (www.w3.org).

You may find it more convenient to use special XML classes rather than working directly with the DOM interfaces. These are described in:

- Working with XML components (see page 2354)
- Abstracting XML documents with the Data Binding wizard (see page 2356)

See Also

Using XML in Database Applications (see page 1847)

3.2.5.7.3 Working with XML Components

The VCL defines a number of classes and interfaces for working with XML documents. These simplify the process of loading, editing, and saving XML documents.

To use the XML classes for examining or editing an XML document you start by setting up an instance of TXMLDocument (see page 2354). You can then work with the nodes (see page 2355) of the XML document component to examine or edit the body of the XML document.

See Also

Using XML in Database Applications (see page 1847)

Using the Document Object Model (see page 2353)

3.2.5.7.4 Using TXMLDocument

The starting point for working with an XML document is the TXMLDocument component.

The following steps describe how to use TXMLDocument to work directly with an XML document:

1. Add a *TXMLDocument* component to your form or data module. *TXMLDocument* appears on the Internet category of the **Tool Palette**.
2. Set the DOMVendor property to specify the DOM implementation you want the component to use for parsing and editing an XML document. The **Object Inspector** lists all the currently registered DOM vendors. For information on DOM implementations, see Using the Document Object Model (see page 2353).
3. Depending on your implementation, you may want to set the ParseOptions property to configure how the underlying DOM implementation parses the XML document.
4. If you are working with an existing XML document, specify the document:

- If the XML document is stored in a file, set the FileName property to the name of that file.
- You can specify the XML document as a string instead by using the XML property.

5. Set the Active property to *True*.

Once you have an active *TXMLDocument* object, you can traverse the hierarchy of its nodes, reading or setting their values. The root node of this hierarchy is available as the DocumentElement property.

For information on working with the nodes of the XML document, see Working with XML nodes (see page 2355).

See Also

Using the Document Object Model (see page 2353)

3.2.5.7.5 Working with XML Nodes

Once an XML document has been parsed by a DOM implementation, the data it represents is available as a hierarchy of nodes. Each node corresponds to a tagged element in the document. For example, given the following XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE StockHoldings SYSTEM "sth.dtd">
<StockHoldings>
   <Stock exchange="NASDAQ">
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
   <Stock exchange="NYSE">
      <name>Pfizer</name>
      <price>42.75</price>
      <symbol>PFE</symbol>
      <shares type="preferred">25</shares>
   </Stock>
</StockHoldings>
```

TXMLDocument would generate a hierarchy of nodes as follows: The root of the hierarchy would be the *StockHoldings* node. *StockHoldings* would have two child nodes, which correspond to the two *Stock* tags. Each of these two child nodes would have four child nodes of its own (*name*, *price*, *symbol*, and *shares*). Those four child nodes would act as leaf nodes. The text they contain would appear as the value of each of the leaf nodes.

Note: This division into nodes differs slightly from the way a DOM implementation generates nodes for an XML document. In particular, a DOM parser treats all tagged elements as internal nodes. Additional nodes (of type text node) are created for the values of the *name*, *price*, *symbol*, and *shares* nodes. These text nodes then appear as the children of the *name*, *price*, *symbol*, and *shares* nodes.

Each node is accessed through an IXMLNode interface, starting with the root node, which is the value of the XML document component's DocumentElement property.

Working with a node's value

Given an *IXMLNode* interface, you can check whether it represents an internal node or a leaf node by checking the IsTextElement property.

- If it represents a leaf node, you can read or set its value using the Text property.
- If it represents an internal node, you can access its child nodes using the ChildNodes property.

Thus, for example, using the XML document above, you can read the price of Borland's stock as follows:

```
BorlandStock := XMLDocument1.DocumentElement.ChildNodes[0];
Price := BorlandStock.ChildNodes['price'].Text;
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes->GetNode(0);
AnsiString Price = BorlandStock->ChildNodes->Nodes[WideString("price")]->Text;
```

Working with a node's attributes

If the node includes any attributes, you can work with them using the Attributes property. You can read or change an attribute value by specifying an existing attribute name. You can add new attributes by specifying a new attribute name when you set the *Attributes* property:

```
BorlandStock := XMLDocument1.DocumentElement.ChildNodes[0];
```

```
BorlandStock.ChildNodes['shares'].Attributes['type'] := 'common';
_di_IXMLNode BorlandStock = XMLDocument1->DocumentElement->ChildNodes->GetNode(0);
BorlandStock->ChildNodes->Nodes[WideString("shares")]->Attributes[WideString("type")] =
"common";
```

Adding and deleting child nodes

You can add child nodes using the *AddChild* method. AddChild creates new nodes that correspond to tagged elements in the XML document. Such nodes are called element nodes.

To create a new element node, specify the name that appears in the new tag and, optionally, the position where the new node should appear. For example, the following code adds a new stock listing to the document above:

```
var
  NewStock: IXMLNode;
  ValueNode: IXMLNode;
begin
  NewStock := XMLDocument1.DocumentElement.AddChild('stock');
  NewStock.Attributes['exchange'] := 'NASDAQ';
  ValueNode := NewStock.AddChild('name');
  ValueNode.Text := 'Cisco Systems'
  ValueNode := NewStock.AddChild('price');
  ValueNode.Text := '62.375';
  ValueNode := NewStock.AddChild('symbol');
  ValueNode.Text := 'CSCO';
  ValueNode := NewStock.AddChild('shares');
  ValueNode.Text := '25';
end;
_di_IXMLNode NewStock = XMLDocument1->DocumentElement->AddChild(WideString("stock"));
NewStock->Attributes[WideString("exchange")] = "NASDAQ";
_di_IXMLNode ValueNode = NewStock->AddChild(WideString("name"));
ValueNode->Text = WideString("Cisco Systems");
ValueNode = NewStock->AddChild(WideString("price"));
ValueNode->Text = WideString("62.375");
ValueNode = NewStock->AddChild(WideString("symbol"));
ValueNode->Text = WideString("CSCO");
ValueNode = NewStock->AddChild(WideString("shares"));
ValueNode->Text = WideString("25");
```

An overloaded version of *AddChild* lets you specify the namespace URI in which the tag name is defined.

You can delete child nodes using the methods of the *ChildNodes* property. ChildNodes is an *IXMLNodeList* interface, which manages the children of a node. You can use its Delete method to delete a single child node that is identified by position or by name. For example, the following code deletes the last stock listed in the document above:

```
StockList := XMLDocument1.DocumentElement;
StockList.ChildNodes.Delete(StockList.ChildNodes.Count - 1);
_di_IXMLNode StockList = XMLDocument1->DocumentElement;
StockList->ChildNodes->Delete(StockList->ChildNodes->Count - 1);
```

See Also

Using TXMLDocument (see page 2354)

Using the Document Object Model (see page 2353)

3.2.5.7.6 Abstracting XML Documents with the Data Binding Wizard

It is possible to work with an XML document using only the *TXMLDocument* component and the *IXMLNode* interface it surfaces for the nodes in that document, or even to work exclusively with the DOM interfaces (avoiding even *TXMLDocument*). However, you can write code that is much simpler and more readable by using the XML Data Binding wizard.

The Data Binding wizard takes an XML schema or data file and generates a set of interfaces that map on top of it. For example, given XML data that looks like the following:

```
<customer id=1>
  <name>Mark</name>
  <phone>(831) 431-1000</phone>
</customer>
```

The Data Binding wizard generates the following interface (along with a class to implement it):

```
ICustomer = interface(IXMLNode)
        ['{8CD6A6E8-24FC-426F-9718-455F0C507C8E}']
        { Property Accessors }
        function Get_Id: Integer;
        function Get_Name: WideString;
        function Get_Phone: WideString;
        procedure Set_Id(Value: Integer);
        procedure Set_Name(Value: WideString);
        procedure Set_Phone(Value: WideString);
        { Methods & Properties }
        property Id: Integer read Get_Id write Set_Id;
        property Name: WideString read Get_Name write Set_Name;
        property Phone: WideString read Get_Phone write Set_Phone;
end;
__interface INTERFACE_UUID("{F3729105-3DD0-1234-80e0-22A04FE7B451}") ICustomer :
    public IXMLNode
{
public:
  virtual int __fastcall Getid(void) = 0 ;
  virtual DOMString __fastcall Getname(void) = 0 ;
  virtual DOMString __fastcall Getphone(void) = 0 ;
  virtual void __fastcall Setid(int Value)= 0 ;
  virtual void __fastcall Setname(DOMString Value)= 0 ;
  virtual void __fastcall Setphone(DOMString Value)= 0 ;
  __property int id = {read=Getid, write=Setid};
  __property DOMString name = {read=Getname, write=Setname};
  __property DOMString phone = {read=Getphone, write=Setphone};
};
```

Every child node is mapped to a property whose name matches the tag name of the child node and whose value is the interface of the child node (if the child is an internal node) or the value of the child node (for leaf nodes). Every node attribute is also mapped to a property, where the property name is the attribute name and the property value is the attribute value.

In addition to creating interfaces (and implementation classes) for each tagged element in the XML document, the wizard creates global functions for obtaining the interface to the root node. For example, if the XML above came from a document whose root node had the tag <Customers>, the Data Binding wizard would create the following global routines:

```
function Getcustomers(Doc: IXMLDocument): IXMLCustomerType;
function Loadcustomers(const FileName: WideString): IXMLCustomerType;
function Newcustomers: IXMLCustomerType;
extern PACKAGE _di_ICustomers __fastcall GetCustomers(TXMLDocument *XMLDoc);
extern PACKAGE _di_ICustomers __fastcall GetCustomers(_di_IXMLDocument XMLDoc);
extern PACKAGE _di_ICustomers __fastcall LoadCustomers(const WideString FileName);
extern PACKAGE _di_ICustomers __fastcall NewCustomers(void);
```

The Get... function takes the interface for a *TXMLDocument* instance . The Load... function dynamically creates a *TXMLDocument* instance and loads the specified XML file as its value before returning an interface pointer. The New... function creates a new (empty) *TXMLDocument* instance and returns the interface to the root node.

Using the generated interfaces simplifies your code, because they reflect the structure of the XML document more directly. For example, instead of writing code such as the following:

```
CustIntf := XMLDocument1.DocumentElement;
CustName := CustIntf.ChildNodes[0].ChildNodes['name'].Value;
_di_IXMLNode CustIntf = XMLDocument1->DocumentElement;
CustName =
CustIntf->ChildNodes->Nodes->GetNode(0)->ChildNodes->Nodes[WideString("name")]->Value;
```

Your code would look as follows:

```
CustIntf := GetCustomers(XMLDocument1);
CustName := CustIntf[0].Name;
_di_ICustomers CustIntf = GetCustomers(XMLDocument1);
CustName = CustIntf->Nodes->GetNode(0)->Name;
```

Note that the interfaces generated by the Data Binding wizard all descend from *IXMLNode*. This means you can still add and delete child nodes in the same way as when you do not use the Data Binding wizard. (See the Adding and deleting child nodes section of Working with XML Nodes (see page 2355).) In addition, when child nodes represent repeating elements (when all of the children of a node are of the same type), the parent node is given two methods, *Add*, and *Insert*, for adding additional repeats. These methods are simpler than using *AddChild*, because you do not need to specify the type of node to create.

The following topics provide detailed information on using the XML Data Binding wizard:

- Using the XML Data Binding wizard (see page 2358)
- Using code that the XML Data Binding wizard generates (see page 2359)

See Also

Using XML in Database Applications (see page 1847)

Using the Document Object Model (see page 2353)

Working with XML Components (see page 2354)

3.2.5.7.7 Using the XML Data Binding Wizard

To use the Data Binding wizard:

1. Choose **File** ▸ **New** ▸ **Other** and select the icon labeled XML Data Binding from the right pane of the New folder located under Delphi Projects.
2. The XML Data Binding wizard appears.
3. On the first page of the wizard, specify the XML document or schema for which you want to generate interfaces. This can be a sample XML document, a Document Type Definition (.dtd) file, a Reduced XML Data (.xdr) file, or an XML schema (.xsd) file.
4. Click the Options button to specify the naming strategies you want the wizard to use when generating interfaces and implementation classes and the default mapping of types defined in the schema to native Delphi data types.
5. Move to the second page of the wizard. This page lets you provide detailed information about every node type in the document or schema. At the left is a tree view that shows all of the node types in the document. For complex nodes (nodes that have children), the tree view can be expanded to display the child elements. When you select a node in this tree view, the right-hand side of the dialog displays information about that node and lets you specify how you want the wizard to treat that node.

- The Source Name control displays the name of the node type in the XML schema.
- The Source Datatype control displays the type of the node's value, as specified in the XML schema.
- The Documentation control lets you add comments to the schema describing the use or purpose of the node.
- If the wizard generates code for the selected node (that is, if it is a complex type for which the wizard generates an interface and implementation class, or if it is one of the child elements of a complex type for which the wizard generates a property on the complex type's interface), you can use the Generate Binding check box to specify whether you want the wizard to generate code for the node. If you uncheck Generate Binding, the wizard does not generate the interface or implementation class for a complex type, or does not create a property in the parent interface for a child element or attribute.
- The Binding Options section lets you influence the code that the wizard generates for the selected element. For any node, you can specify the Identifier Name (the name of the generated interface or property). In addition, for interfaces, you must indicate which one represents the root node of the document. For nodes that represent properties, you can specify the type of the property and, if the property is not an interface, whether it is a read-only property.

6. Once you have specified what code you want the wizard to generate for each node, move to the third page. This page lets you choose some global options about how the wizard generates its code and lets you preview the code that will be generated, and lets you tell the wizard how to save your choices for future use.

- To preview the code the wizard generates, select an interface in the Binding Summary list and view the resulting interface definition in the Code Preview control.
- Use the Data Binding Settings to indicate how the wizard should save your choices. You can store the settings as annotations in a schema file that is associated with the document (the schema file specified on the first page of the dialog), or you can name an independent schema file that is used only by the wizard.

7. When you click Finish, the Data Binding wizard generates a new unit that defines interfaces and implementation classes for all of the node types in your XML document. In addition, it creates a global function that takes a *TXMLDocument* object and returns the interface for the root node of the data hierarchy.

See Also

Using Code That the XML Data Binding Wizard Generates (see page 2359)

3.2.5.7.8 Using Code That the XML Data Binding Wizard Generates

Once the wizard has generated a set of interfaces and implementation classes, you can use them to work with XML documents that match the structure of the document or schema you supplied to the wizard. Just as when you are using only the built-in XML components (see page 2354), your starting point is the TXMLDocument component that appears on the Internet category of the **Tool Palette**.

To work with an XML document, use the following steps:

1. Obtain an interface for the root node of your XML document. You can do this in one of three ways:

Method	Description
Place a *TXMLDocument* component in your form or data module. Bind the *TXMLDocument* to an XML document by setting the FileName property.	(As an alternative approach, you can use a string of XML by setting the XML property at runtime.) Then, In your code, call the global function that the wizard created to obtain an interface for the root node of the XML document. For example, if the root element of the XML document was the tag <StockList>, by default, the wizard generates a function *Getstocklist*, which returns an *IXMLStockListType*. interface: var StockList: IXMLStockListType; begin XMLDocument1.FileName := 'Stocks.xml'; StockList := Getstocklist(XMLDocument1); XMLDocument1->FileName := "Stocks.xml"; _di_IStockListType StockList = GetStockListType(XMLDocument1);
Call the generated Load... function	Call the generated Load... function to create and bind the *TXMLDocument* instance and obtain its interface all in one step. For example, using the same XML document described above: var StockList: IXMLStockListType; begin StockList := Loadstocklist('Stocks.xml'); _di_IStockListType StockList = LoadStockListType("Stocks.xml");
Call the generated New... function	Call the generated New... function to create the *TXMLDocument* instance for an empty document when you want to create all the data in your application: var StockList: IXMLStockListType; begin StockList := Newstocklist; _di_IStockListType StockList = NewStockListType();

2. This interface has properties that correspond to the subnodes of the document's root element, as well as properties that correspond to that root element's attributes. You can use these to traverse the hierarchy of the XML document, modify the data in the document, and so on.
3. To save any changes you make using the interfaces generated by the wizard, call the *TXMLDocument* component's SaveToFile method or read its XML property.

   Tip: If you set the *Options* property of the *TXMLDocument* object to include *doAutoSave*, then you do not need to explicitly call the *SaveToFile* method.

See Also

Using the XML Data Binding Wizard (see page 2358)

Index

#

(null Directive) 711
character 695
#define 690
#error 692
#if 688
#if, #elif, #else, And #endif 688
#ifdef 688
#ifdef And #ifndef 688
#include 692
#line 693
#pragma 696
#pragma alignment 698
#pragma anon_struct 698
#pragma argsused 699
#pragma checkoption 699
#pragma codeseg 699
#pragma comment 699
#pragma comments, pragma 699
#pragma defineonoption 700
#pragma defineonoption and #pragma undefineonoption 700
#pragma exit 700
#pragma exit and #pragma startup 700
#pragma hdrfile 700
#pragma hdrstop 701
#pragma inline 701
#pragma intrinsic 701
#pragma link 701
#pragma message 702
#pragma nopushoptwarn 702
#pragma obsolete 702
#pragma option 703
#pragma pack 704
#pragma package 706
#pragma region 707
#pragma region and #pragma end_region 707
#pragma resource 707
#pragma warn 707
#undef 691

*

* operator
    operators, C++ 658

.

. (direct Member Selector) 642
. operator
    operators, C++ 642

?

?: operator
    operators, C++ 592

[

[ ] operator
    operators, C++ 643

_

__alignment_of 473
__array_extent 473
__array_rank 473
__automated 522
    keywords, C++ 522
__classid 523
    keywords, C++ 523
__classmethod 574
    keywords, C++ 574
__closure 523
    keywords, C++ 523
__declspec 523
__declspec(dllexport) 540
__declspec(dllimport) 541
__declspec(naked) 541
__declspec(noreturn) 541

__declspec(nothrow) 542
__declspec(novtable) 542
__declspec(property) 543
__declspec(selectany) 543
__declspec(thread) 544
__declspec(uuid(“ComObjectGUID”)) 544
__declspec,
keywords, C++ 523
__declspec, dllexport
keywords, C++ 540
__declspec, dllimport
keywords, C++ 541
__declspec, naked
keywords, C++ 541
__declspec, noreturndeclspec, noreturn
keywords, C++ 541
__declspec, nothrow
keywords, C++ 542
__declspec, novtable
keywords, C++ 542
__declspec, property
keywords, C++ 543
__declspec, selectany
keywords, C++ 543
__declspec, thread
keywords, C++ 544
__declspec, uuid
keywords, C++ 544
__dispid 525
__except 525
keywords, C++ 525
__finally 550
__finally keyword 2023
__finally,
keywords, C++ 550
__FUnloadDelayLoadedDLL 790
__has_nothrow_assign 473
__has_nothrow_copy_constructor 474
__has_nothrow_default_constructor 474
__has_trivial_assign 475
__has_trivial_copy_constructor 475
__has_trivial_default_constructor 476
__has_trivial_destructor 477
__has_virtual_destructor 478
__import
keywords, C++ 554
__inline 525
__int8, __int16, __int32, __int64, Unsigned __int64, Extended Integer Types 526
__int8, __int16, and so forth
lexical elements, C++ 526
__is_abstract 478
__is_arithmetic 478
__is_array 479
__is_base_of 479
__is_class 479
__is_complete_type (typename T) 480
__is_compound 480
__is_const 481
__is_convertible 481
__is_empty 481
__is_enum 482
__is_floating_point 482
__is_function 483
__is_fundamental 483
__is_integral 484
__is_lvalue_expr 484
__is_lvalue_reference 484
__is_member_function_pointer 485
__is_member_object_pointer 485
__is_member_pointer 485
__is_object 485
__is_pod 486
__is_pointer 486
__is_polymorphic 487
__is_reference 487
__is_rvalue_expr 487
__is_rvalue_reference 488
__is_same 488

__is_scalar 488
__is_signed 489
__is_standard_layout 489
__is_trivial 490
__is_union 491
__is_unsigned (typename T) 491
__is_void 491
__is_volatile 491
__msfastcall 526
    keywords, C++ 526
__msreturn 527
    keywords, C++ 527
__pfnDliNotifyHook, __pfnDliFailureHook 787
__property 527
__published 528
__rtti
    C++ language specifics 528
__rtti, -RT Option 528
__thread, Multithread Variables 529
__throwExceptionName 840
__throwFileName 840
__throwLineNumber 840
__try 529
    keywords, C++ 529
_adopt_thread 975
_argc 1095
_argv, _wargv 1095
_atoi64, _wtoi64 1096
_atold, _wtold 933
_beginthread 976
_beginthreadex 980
_beginthreadNT 978
_Bool 578
    keywords, C++ 578
_bool __has_trivial_destructor (typename T)
    C++0x type trait functions 477
_c_exit 981
_cexit 982
_chdrive 791
_chgsign, _chgsignl 856
_clear87, _clearfp 857
_Complex 578
    keywords, C++ 578
_control87, _controlfp 858
_copysign, _copysignl 859
_creat, _wcreat 892
_crotl, _crotr 1096
_crotr, _crotl 1096
_ctype 772
_daylight 1202
_doserrno 833
_endthread 983
_endthreadex 984
_environ, _wenviron 1113
_exit 1097
_expand 984
_export
    keywords, C++ 530
_export, __export 530
_F_xxxx #defines 1023
_fastcall
    keywords, C++ 530, 550, 557
_fastcall, __fastcall 530
_fcloseall 1031
_fdopen, _wfdopen 1032
_fgetchar, _fgetwchar 1037
_fileno 1040
_findclose 873
_findfirst, __wfindfirst 874
_findfirsti64, _wfindfirsti64 875
_findnext, __wfindnext 875
_findnexti64, _wfindnexti64 876
_finite, _finitel 859
_flushall 1040
_fmode 847
_fpclass, _fpclassl 860
_fpreset 860
_fputchar, _fputwchar 1044

_fsopen, _wfsopen 1025
_fullpath, _wfullpath 1098
_get_osfhandle 877
_getdcwd, _wgetdcwd 791
_getdrive 822
_getw 1054
_heapchk 716
_heapmin 717
_heapset 717
_i64toa, _i64tow 934
_Imaginary 578
    keywords, C++ 578
_IOxxx #defines 1025
_IS_xxx #defines 772
_ismbblead, _ismbbtrail 1152
_ismbclegal 1153
_ismbslead, _ismbstrail 1153
_isnan, _isnanl 861
_itow, itoa 1117
_logb, _logbl 862
_lrand 1099
_lrotl, _lrotr 1099
_lrotr, _lrotl 1099
_ltoa, _ltow, ltoa 1121
_makepath, _wmakepath 1100
_matherr, _matherrl 935
_mbbtype 1154
_mbccpy 1154
_mbsbtype 1154
_mbslen, strlen, wcslen, _mbstrlen 1131
_mbsnbcmp 1155
_mbsnbcnt, _mbsnccnt, _strncnt, _wcsncnt 1155
_mbsnbcoll, _mbsnbicoll 1156
_mbsnbcpy 1157
_mbsnbicmp 1157
_mbsnbset 1158
_mbsninc, _strninc, _wcsninc 1158
_mbsspnp, _strspnp, _wcsspnp 1158
_mktemp, _wmktemp 815
_msize 720
_new_handler 969
_nextafter, _nextafterl 862
_open_osfhandle 879
_osmajor 823
_osminor 823
_osversion 823
_pclose 1027
_pipe 847
_popen, _wpopen 1027
_putw 1059
_rmdir, _wrmdir 816
_rmtmp 1063
_rotl, _rotr 1102
_rotr, _rotl 1102
_rtl_chmod, _wrtl_chmod 880
_rtl_close 881
_rtl_creat, _wrtl_creat 882
_rtl_heapwalk 720
_rtl_open, _wrtl_open 883
_rtl_read 885
_rtl_write 886
_scalb, _scalbl 863
_searchenv, _wsearchenv 1103
_searchstr, _wsearchstr 1104
_setcursortype 740
_sleep 824
_snprintf;_snwprintf 1028
_sopen, _wsopen 850
_splitpath, _wsplitpath 1105
_stati64, _tstati64, stati64, _wstati64 1188
_status87, _statusfp 863
_stdcall
    keywords, C++ 531
_stdcall, __stdcall 531
_strdate, _wstrdate 1197
_strdec, mbsdec, _wcsdec 1159
_strerror 1159
_strinc, mbsinc, _wcsinc 1160

_strnextc,_mbsnextc,_wcsnextc 1161
_strtime, _wstrtime 1198
_sys_errlist 837
_sys_nerr 838
_TCHAR maps to (C++ option) 501
_tempnam, _wtempnam 1071
_TEXT function 501
_threadid 1005
_timezone 1209
_tolower 772
_toupper 773
_tzname,_wtzname 1210
_tzset, _wtzset 1210
_ui64toa, _ui64tow 1106
_ui64tow, _ui64toa 1106
_ultow, ultoa 1137
_unadopt_thread 985
_unlink, _wunlink 830
_unsigned __int_alignment_of(typename T)
    C++0x type trait functions 473
_utime, _wutime 1213
_version 824
_vsnprintf;_vsnwprintf 1029
_wargv, _argv 1095
_wcstod, wcstod 1131
_wenviron, _environ 1113
_wgetenv, getenv 1116
_wputenv, putenv 1126
_wsearchenv,, _searchenv 1103
_wsearchstr, _searchstr 1104
_wsplitpath, _splitpath 1105
_wsystem, system 1136
_wtoi, atoi
    conversion routine 1108
_wtoi64, _atoi64 1096
_wtol, atol 1109

+

+ operator
    operators, C++ 657
++ operator
    operators, C++ 643

<

<name> is not a valid identifier (C++) 361

-

-> (indirect Member Selector) 643
>, <, >=, <= Relational Operators 591
Abnormal program termination (C++) 286
abort 1107
abort And Destructors 425
About Crosstabs 1804
About The main() Function 627
About Web Broker and WebSnap 2246
abs 936
Abstract Class Members 1370
abstract classes
    C++ language specifics 446
Abstract Classes 446
abstract methods 1242
    methods 1370
Abstract Methods 1242
Abstracting XML Documents with the Data Binding Wizard 2356
Accepting Dragged Items 2153
Access Errors 4
Access Methods (properties) 1252
access specifiers
    C++ language specifics 399
access, _waccess 887
accessing
    namespaces, C++ 427, 428, 429, 572
Accessing a Data Module from a Form 1910
Accessing a Particular String 2134
Accessing client request information 2273
Accessing Client Request Information 2274
Accessing dbExpress Schema Information 1824
Accessing Elements Of A namespace 427

Accessing Field Values with a Dataset's FieldByName Method 1853
Accessing Field Values with a Dataset's Fields Property 1854
Accessing Field Values with the Default Dataset Property 1872
Accessing Schema Information 51
Accessing the Connection Object 1475
Accessing the Connection's Datasets 1475
Accessing the Day, Month, and Year 1260
Accommodating Varying Color Depths 1945
acos, acosl 937
action bands
creating dynamic menus 1977
creating most recently used lists 1977
hiding unused items 1986
action editor
action items 2277
action items 2278
default 2278
enabling 2279
HTTP requests 2280
properties 2279
Action items 2277
Action Items 2278
action lists
actions 2003
actions 2011
action bands 1978, 2002
action lists 1981, 1992, 1996
component writing using 1987, 2013
executing 1984
predefined 1995
using 2007
Activating a Session 1651
active documents 1394
Active Documents 1394
Active Forms
client applications 1526
Active Server Objects
out-of-process servers 1405
registering 1405
testing and debugging 1406
Active server page
overview 1400
Active Server Page
creating 1401
intrinsic objects 1402
Active Server Pages 1393
overview 1393
ActiveX applications
registering controls 139
ActiveX buttons
creating in a VCL form 139
ActiveX controls
adding additional properties 1412
connecting with property page 1417
creating a property page 1415
customizing 1411
deploying 1945
description 1394
designing 1409
elements of 1408
for Web deployment 1410
licensing 1410
ActiveX Controls 1394
Adapter dispatcher operation 2316
Adapter Dispatcher Operation 2317
Adapters 2325
Adding a CoClass to the Type Library 1463
Adding a Cool Bar Component 1968
Adding a Field to a Form Object to Track Mouse Actions 2181
Adding a Module to the Type Library 1465
Adding a New Connection to the Data Explorer 73
Adding a New Index 1709
Adding a Record or Union to the Type Library 1464
Adding a Remote Data Module to an Application Server Project 1910
Adding a Session to the Web Module 2264
Adding a Speed Button to a Panel 1969
Adding a String to a List 2134
Adding a Tool Button 1971

Adding a Toolbar Using a Panel Component 1969
Adding a Toolbar Using the Toolbar Component 1970
Adding Actions to the Dispatcher 2277
Adding Additional Properties, Methods, and Events 1412
Adding an Action to the Action List 1278
Adding an Alias to the Type Library 1464
Adding an Enumeration to the Type Library 1464
Adding an Image Control 2181
Adding an Image to the Image List 1279
Adding an Interface to a CoClass 1463
Adding an Interface to the Type Library 1461
Adding and Sorting Strings 138
Adding Application-specific Information to the Data 1709
Adding Audio and/or Video Clips to an Application 2177
Adding Clipboard Formats 1344
Adding Color, Patterns, or Pictures to Menus, Buttons, and Toolbars 1966
Adding Commands to the Ribbon 144
Adding Component Editors 1344
Adding Controls to a Property Page 1416
Adding Custom Components to the Tool Palette 2144
Adding Custom Information to Data Packets 1808
Adding Forms 1965
Adding Graphic Capabilities 1219
Adding Graphical Objects to a String List 2154
Adding Graphics to Controls 2154
Adding Hidden Toolbars 1971
Adding Icons to Menus and Toolbars 1966
Adding Images to a String List 2154
Adding Images to an Application 2155
Adding Images to Menu Items 1967
Adding Interface Properties 1359
Adding Items to the Context Menu 1345
Adding Items to the Object Repository 1915
Adding Login Support 2328
Adding New Capabilities to a Class 1370
Adding New Measurement Types 2091
Adding New Records 1583
Adding New Web Services 2301
Adding Properties and Methods to the Type Library 1462
Adding Property Editors 1345
Adding Rave Reports to RAD Studio 81
Adding Scroll Bars at Runtime 2156
Adding Silent Video Clips to an Application 2179
Adding the Clipboard Object 2156
Adding the Data Link 1327
Adding the Execute Method 1359
Adding the ReadOnly property 1328
Adding, Inserting, and Deleting Menu Items 1968
Additional Methods 1928
ADO
    asynchronous fetching 1481
    batch updates 1491
    connecting to data stores 1479
    connection components 98
    connection modes 1485
    connection object 1475
    recordset objects 1494
ADO command components
    SQL commands 1492
ADO components
    databases 1494
ADO connection components
    associated commands 1475
    connections 1478, 1483
ADO Connection Events 1474
ADO connections
    events 1474
ADO datasets 1490, 1492
ADT fields 1864
Advanced HTML Design 2311
Advantages of the Multi-tiered Database Model 1519
aggregate fields
    defining 1856
Aggregating over groups of records 1710
aggregation
    COM objects 1391
    interfaces 2074
Aggregation 2074

Aggregation (COM) 1391
alias
    namespaces, C++ 429
alignas 575
    keywords, C++ 575
alignof 531
    keywords, C++ 531
alignof Operator (C++0x) 463
alloc.h 712
alloca 920
Allowing Needed Updates 1329
Allowing Toggle Buttons 1971
Allowing Toggled Tool Buttons 1972
Alternative Representations of Tokens 579
Analyzing Data 1567
ancestor classes 1370
    descendant classes 1371
ancestor component 97
Ancestors, Descendants, and Class Hierarchies 1371
and
    alternative representations of C++ tokens 531
and, && 531
and_eq, &= 575
animation
    AVI 2037
Animation Control 2037
anonymous methods
    C++ handling 508
anonymous namespaces
    namespaces, C++ 427
Anonymous namespaces 427
Anonymous Unions 659
AnsiStringT and code page
    C++ handling 510
application
    events 99, 101, 106, 109, 112, 123
    VCL Forms 101
application files
    file name extensions 1945
    identifying 1950
Application Files, Listed by File Name Extension 1945
Application is running (C++) 361
application servers
    access and launch permissions 1532
    creating 1527
    providing data 1816
    registering 1535
    structure 1542
    Web clients 1521
applications
    compiling 1937
    console 1900
    creating 1883, 1902, 1936
    debugging 1937
    deploying 1937, 1947
    designing 1935
Apply Updates Dialog 1466
Applying BDE-based Cached Updates 1691
Applying Cached Updates Using a Database 1687
Applying Cached Updates with Dataset Component Methods 1695
Applying or Canceling a Range 1585
Applying the Batch Updates to Base Tables 1476
Applying Updates 1711
Applying Updates to Datasets That do Not Represent a Single Table 1809
Approaches to File I/O 2091
architecture
    BDE-based 1638
    database applications 1562
    single-tiered applications 1560, 1564
Arguments to main() 628
arithmetic operators
    operators, C++ 656
Arithmetic Operators 656
Arranging Persistent Fields 1854
Arranging the Order of Persistent Columns 1751
array
    fields 1866

properties 1253
Array Subscript Operator 643
arrays
C++ language specifics 437
language structure, C++ 638
Arrays 638
as operator
operators 2082
asctime 1199
asin, asinl 938
Asking the Help Manager for Information 1890
asm
keywords, C++ 532
asm, _asm, __asm 532
assert 731
assert.h 730
Assigning a Menu to a Tool Button 1972
Assigning a Speed Button's Glyph 1972
Assigning Data Directly 1738
Assigning Images to Tool Buttons 1973
Assigning Separate Names to Similar Threads 2239
assignment
operators, C++ 591
Assignment Operators 591
Assignment To Enum Types 618
Associating a Data Control with a Dataset 1746
Associating a Database Component with a Session 1645
Associating a Dataset with Database and Session Connections 1668
Associating an Event with an Existing Event Handler 2146
Associating Attribute Sets with Field Components 1854
Associating Menu Events with Event Handlers 2144
Associating Objects with a String List 2135
Associating Property Page Controls with ActiveX Control Properties 1416
atan, atanl 939
atan2, atan2l 940
atexit 1107
atexit, #pragma exit, And Destructors 426
atof, _wtof 933
atoi, _wtoi 1108
atol, _wtol 1109
Attributes 580
Attributes noreturn and final (C++0x) 492
audio
video 2177, 2179
auto 533
keywords, C++ 533
Automation
managing events 1437
Servers 1393
Automation controller
dispatch interface 1424
Automation controllers
importing a type library 1421
writing 1423
Automation Interfaces 1438
Automation server
creating 1432
Automation servers
connecting to 1424
debugging 1440
Automation Servers 1393
avi
animation 2179, 2180
Avoiding Interdependencies 1242
Avoiding Simultaneous Access 2226
Avoiding Simultaneous Thread Access to the Same Memory 125
axiom 575
keywords, C++ 575
bad_cast class 1212
bad_typeid class 1212
Base And Derived Class Access 397
base classes
C++ language specifics 397
batch operations
mapping data types 1684
modes 1682
running 1680

batch updates
canceling 1476
BatchMove component
adding 1679
batch operations 1683
error handling 1684
BCC32, the C++ Command-Line Compiler 159
BCC32.EXE 159
BDE
utilities 1681
BDE Overview 15
BDE-based Architecture 1638
BeforeUpdateRecord event
OnGetTableName event 1718
bevels 2037
Bevels 2037
Binary Operators 587
Binding Parameters 1669
Bit Definitions for fnsplit 811
bit fields
language structure, C++ 653
Bit Fields 653
bitand
C++ language specifics 533
bitand, & 533
bitmap buttons 2038
Bitmap Buttons 2038
bitmaps
drawing on 2188
offscreen bitmaps 1379
setting size 2201
bitor
alternative representations of C++ tokens 533
bitor, | 533
BITSPERBYTE #define 1215
Bitwise Operators 589
bitwise operators,
operators, C++ 589
Blackfish SQL
overview 24
Blackfish SQL Overview 24
BLOBs
caching 1670
blocking connections 2340
Blocking Connections 2340
blocks
language structure, C++ 647
Blocks 647
bool
keywords, C++ 533
bool __is_abstract (typename T)
C++0x type trait functions 478
bool __is_empty(typename T)
C++0x type trait functions 481
bool __ has_nothrow_copy_constructor(typename T)
C++0x type trait functions 474
bool __has_nothrow_assign (typename T)
C++0x type trait functions 473
bool __has_nothrow_default_constructor(typename T)
C++0x type trait functions 474
bool __has_trivial_assign(typename T)
C++0x type trait functions 475
bool __has_trivial_default_constructor(typename T)
C++0x type trait functions 476
bool __has_virtual_destructor (typename T)
C++0x type trait functions 478
bool __is_arithmetic(Typename T)
C++0x type trait functions 478
bool __is_array(Typename T)
C++0x type trait functions 479
bool __is_base_of (typename Base, typename Derived)
C++0x type trait functions 479
bool __is_class(T)
C++0x type trait functions 479
bool __is_complete_type(T)
C++0x type trait functions 480
bool __is_compound(typename T)
C++0x type trait functions 480

bool __is_const (typename T)
C++0x type trait functions 481
bool __is_convertible (typename From, typename To)(typename T)
C++0x type trait functions 481
bool __is_enum_type(typename T)
C++0x type trait functions 482
bool __is_floating_point(typename T)
C++0x type trait functions 482
bool __is_function(typename T)
C++0x type trait functions 483
bool __is_fundamental (typename T)
C++0x type trait functions 483
bool __is_integral(typename T)
C++0x type trait functions 484
bool __is_lvalue_expr(typename T)
C++0x type trait functions 484
bool __is_lvalue_reference(typename T)
C++0x type trait functions 484
bool __is_member_function_pointer(typename T)
C++0x type trait functions 485
bool __is_member_object_pointer(typename T)
C++0x type trait functions 485
bool __is_member_pointer(typename T)
C++0x type trait functions 485
bool __is_object(typename T)
C++0x type trait functions 485
bool __is_pod(typename T)
C++0x type trait functions 486
bool __is_pointer(Typename T)
C++0x type trait functions 486
bool __is_polymorphic(typename T)
C++0x type trait functions 487
bool __is_reference(typename T)
C++0x type trait functions 487
bool __is_rvalue_expr(typename T)
C++0x type trait functions 487
bool __is_rvalue_reference(typename T)
C++0x type trait functions 488
bool __is_same (typename T, typename U)
C++0x type trait functions 488
bool __is_scalar(typename T)
C++0x type trait functions 488
bool __is_signed(typename T)
C++0x type trait functions 489
bool __is_standard_layout(typename T)
C++0x type trait functions 489
bool __is_trivial(typename T)
C++0x type trait functions 490
bool __is_union(typename T)
C++0x type trait functions 491
bool __is_unsigned(typename T)
C++0x type trait functions 491
bool __is_void(typename T)
C++0x type trait functions 491
bool __is_volatile(typename T)
C++0x type trait functions 491
bool, false, true 533
bool_has __trivial_default_constructor(typename T)
C++0x type trait functions 475
Borland Database Engine 1946
aliases 1648, 1657
deploying 1946
direct calls 1671
BRC32, the Resource Shell 163
BRC32.EXE 163
BRCC32.EXE 165
BRCC32.EXE, the Resource Compiler 165
break 534
keywords, C++ 534
briefcase model
mobile computing 1565
Broadcast method
sending messages 1301
Broadcasting a Message to All Controls in a Form 1301
Brokering Connections 1520
browsing a database 52, 74
Browsing a Database in the Data Explorer 74
brush bitmap property 2201

brush color 2181
brushes 2205
bsearch 1110
BUFSIZ #define 1023
Building a "Hello World" Web Services Application 153
Building a Multithreaded Application 124
Building a Multi-tiered Application 1520
Building a VCL Forms "Hello World" Application 101
Building a VCL Forms ADO Database Application 98
Building a VCL Forms Application 99
Building a VCL Forms Application with Decision Support Components 91
Building a VCL Forms dbExpress Database Application 103
Building a VCL Forms MDI Application Using a Wizard 93
Building a VCL Forms MDI Application Without Using a Wizard 94
Building a VCL Forms SDI Application 96
Building a VCL Forms Web Browser Application 142
Building a WebSnap "Hello World" Application 149
Building a WebSnap Application 148
Building a Windows "Hello World" Console Application 88
Building an Application with XML Components 104
Building an InternetExpress Application 1521
Building Application Menus 90
Building applications, components, and libraries 1879
Building Menus 1973
Building VCL Forms Applications With Graphics 93
Building Web Applications Using InternetExpress 1522
Building with MAKE 183
button control
    event handling 88
button controls 2038
Button Controls 2038
Buttons and Similar Controls 2038
C compiler preprocessor 167
C Runtime Library Reference 711
C++ Classes 387
C++ compiler
    command line 159
    conversion via user-defined operators 459
C++ Compiler Errors And Warnings (C++) 216
C++ compiler strictness
    binding references and qualifiers 455
    function overload resolution 458
    initialization and conversion 459
    string literals 457
    template changes 457
C++ import library conversion tool 166
C++ incremental linker 174
C++ language
    C++ language specifics 647
C++ Language Guide 385
C++ namespaces 426
C++ Reference 156
C++ Reference Declarations 639
C++ resource compiler 165
C++ resource shell 163
C++ Scope 429, 430
C++ Scoping Rules Summary 431
C++ Specific Keywords 581
C++ Specific Operators 590
C++ Specifics 385, 647
C++0x features
    alignof operator 463, 495
    attributes noreturn and final 492
    decltype specifier 499
    explicit conversion operatorsl 493
    extern templates 494
    forward declaration of enums 495
    static assertions 498
    strongly typed enums 498
    type trait functions 472
    Unicode char types 500
C++0x Features (C++Builder 2009) 462
C++Builder Keyword Extensions 580
C++-Specific Keywords 681
cabs, cabsl 940
cached updates
    applying 1691

BDE 1689
BDE-based 1688
error handling 1696
caching
web page 148
Caching BLOBs 1670
calculated fields 1585
assigning values 1870
defining 1872
Calculating Fields 1585
Calling a Control's Message Handler Directly 1302
Calling a Help System Directly 1890
Calling Invokable Interfaces 2307
Calling Methods 2145
Calling Server Interfaces 1522
Calling the Apply Method 1642
Calling the Click-event Handler 1235
Calling the Event 1240
calloc 721
Canceling Batch Updates 1476
Canceling Changes 1586
Canceling Commands 1477
canceling data changes 1586
canvas
properties and methods 2184
canvas methods
graphic objects 2205
canvas object
properties 2207
canvases
graphics 1381
case 535
keywords, C++ 535
casting, C++ style
C++ language specifics 438
catch 536
keywords, C++ 536
cdecl
keywords, C++ 536
cdecl, _cdecl, __cdecl 536
ceil, ceill 942
cgets 741
Chaining Page Producers Together 2268
Change method 1334
ChangeScale
typecasts 2169
Changing Class Defaults to Avoid Repetition 1371
Changing Initial Values 1262
Changing Message Handling 1304
Changing Other Decision Graph Series Properties 1786
Changing Other Decision Graph Template Properties 1786
Changing the Associated Dataset at Runtime 1747
Changing the Brush Color 2181
Changing the Brush Style 2175
Changing the Default Decision Graph Type 1786
Changing the Default Value of FReadOnly 1330
Changing the Double-click Behavior 1345
Changing the Drag Mouse Pointer 2157
Changing the Name of a Component 2065
Changing the Pen Color 2182
Changing the Pen Mode 2182
Changing the Pen Style 2182
Changing the Pen Width 2183
Changing the Series Graph Type 1787
Changing the Standard Event Handling 1238
Changing the Tool with Speed Buttons 2184
Changing VCL for the Web application type 2257
char 536
keywords, C++ 536
CHAR_xxx #defines 913
char16_t 537
keywords, C++ 537
char32_t 537
keywords, C++ 537
Character Constants 663
Character Constants Overview 662
Character Sets 1924
character types

lexical elements, C++ 666
chdir 803
check boxes 2039
data-aware 1775
Check Boxes 2039
Checking for Termination by Other Threads 2232
child nodes 104
child windows
Close event 94
closing all 94
main menu 94
MDI applications 94
chmod, _wchmod 889
Choosing a Connection Protocol 1523
Choosing a Default Action Item 2278
Choosing a Threading Model 1433
Choosing How to Apply Updates Using a Dataset Provider 1810
Choosing How to Organize the Data 1747
Choosing Navigator Buttons to Display 1752
Choosing the Type of Dataset for Caching Updates 1711
chsize 890
class 537
keywords, C++ 537
class factories
COM objects 1388
class initialization
C++ language specifics 417
Class Initialization 417
class library
TObject branch 2058
Class library 1314
Class Member List 390
class members
visibility 2069
class methods
C++ language specifics 406
Class Methods 406
class name
specifying for a component 97
Class Name Scope 393
Class Names 396
Class Objects 402
Class Scope 431
Class Templates 412
Class Templates Overview 411
Class Types 404
classes
defining 2072
defining new 1370
deriving 1318
deriving new 1370
removing dependencies 1321
Classes 386
Classes and Pointers 1374
classes,
C++ language specifics 387
classes, forward
C++ language specifics 405
classes, member list
C++ language specifics 390
classes, names
C++ language specifics 393, 396
classes, objects
C++ language specifics 402
classes, scope
C++ language specifics 431
classes, types
C++ language specifics 404
clearerr 1029
clicks
responding to 1996
client applications
calling server interfaces 1522
connecting to servers 1519
dropping connections 1531
multi-tiered 1543
client connections 2344
Client Connections 2344

client datasets 1710, 1740
adding indexes 1709
architecture 1559
cached updates 1723, 1731
calculated values 1725
constraints 1713
copying data 1714, 1738
creating 1714
data manipulation 1737
data packets 1725
deleting indexes 1715
editing 1715
file-based applications 1733
grouping data 1734
internally calculated fields 1735
limiting records in data packets 1720
maintained aggregates 1721, 1730, 1735
navigating 1721
optional parameters 1709
parameters 1723
providers 1561, 1729
query parameters 1727
saving changes 1726
sharing data 1736
sorting and indexing 1728
types 1711, 1717
undoing changes 1730
using providers 1732
with internal source dataset 1733
Client Events 2342
client sockets
ClientSocket component 2348
events 2342
clients
COM 1391
multi-tiered applications 1528
Clipboard 2156
clock 1200
clock_t 1201
Cloning a Client Dataset Cursor 1736
close 891
closedir, wclosedir 794
Closing Database Connections 1656
Closing Server Connections 2349
Closing the Connection 2347
clreol 742
clrscr 743
CoClasses and Class Factories 1388
code
editing 1936
Code Generated by Wizards 1398
Code Generated When You Import Type Library Information 1421
code sharing among event handlers 2202
CodeGuard
errors 4, 5, 6, 7
overview 9
using 48
warnings 11
CodeGuard Errors 3
CodeGuard Overview 9
CodeGuard Procedures 48
CodeGuard Warnings 10
COFF2OMF.EXE 166
COFF2OMF.EXE, the Import Library Conversion Tool 166
columns
lookup lists 1757
properties 1758, 1762, 1763
COM
clients 1419
exposing properties 1435
extensions 1385
overview 1385
overview of creating objects 1430
specification and implementation 1385
COM application parts 1386
COM applications
DCOM applications 1918
COM basics 1382

COM clients
importing a type library 1420
writing 1422
COM Clients 1391
COM extensions 1392
COM Extensions 1392
COM Interface Pointers 1387
COM interfaces
IUnknown 1387
pointers 1387
COM Interfaces 1386
Interfaces 1386
COM Object Instancing Types 1433
COM objects
choosing a threading model 1433
creating 1431
designing 1430
implementing 1397, 1398
instancing types 1433
marshaling data 1439
registering 1440
COM servers
COM object 1388
types of 1389
COM Servers 1388
COM wizards 79
COM+ 1399
COM+ Event And Event Subscriber Objects 1399
Combining Approaches 1565
combo boxes 2039
Combo Boxes 2039
Comma Operator 592
Command Line Utilities 156
CommandText
client datasets 1722
Comments 685
comments,
lexical elements, C++ 685
Common Controls and XP Themes 1974
common dialog boxes
using 2013
Common List Operations 2091
Common Properties and Methods of Canvas 2184
Commonly Used Long String Routines 2118
Commonly Used Routines for Null-terminated Strings 2121
Communicating with Providers Using Custom Events 1712
Communicating with the Client Dataset 1811
Communicating with the Help Manager 1890
compiler directives
strings 2122
Compiler Directives for Strings 2122
compiler options
project options 1900
compiler options (C++) 159
Compiler Template Switches 413
Compiling and Linking from the Command Line 2222
Compiling Applications 1937
Compiling Components into Packages 1346
Compiling Packages 2212
compl
alternative representations of C++ token 538
compl, ~ 538
component
creating 97
component editors
clipboard formats 1344
context menus 1345
creating 1344
double-clicks 1345
registering 1353
Component wizard
components 1321
component wrappers
COM 1422
Component Writer's Guide 1217
component writing 1365
adding graphic capabilities 1219
controlling access 1371

default property values 1366
drawing the component image 1224
hiding implementation details 1374
making a control read-only 1333
properties 1250
providing an OnChange event 1270
publishing inherited properties 1227
components 1313
adding to the Tool palette 1350
classes 1315
creating 1315, 1317
creating and registering 1363
designing 1323
graphics 1380
grouping 2041
installing 1346, 2144
memory management 2066
properties 1319
property categories 1350, 1351, 1352, 1353, 1356
registering 1320, 1321
renaming 2065
testing 1324
troubleshooting 1356
VCL Forms 89
Components and Classes 1315
Components and Ownership 2066
Components Available Only on Specific OS 39
Composing Client Requests 2248
Composing Update SQL Statements 1687
concept 576
keywords, C++ 576
concept_map 576
keywords, C++ 576
Concepts 1
conditional compilation 687
Conditional Compilation 687
Conditional Compilation Overview 687
Conditional Operators 592
configuring
IIS 98, 103, 149
Configuring TMTSDataModule 1524
Configuring TRemoteDataModule 1525
Configuring TSOAPDataModule 1525
Configuring TSQL Connection 53
conio.h 732
Connecting a Client Dataset to Another Dataset in the Same Application 1559
Connecting a Property Page to an ActiveX Control 1417
Connecting an ADO Dataset to a Data Store 1479
connecting DataGrids 98
Connecting Directly to a Database Server 1560
Connecting to a Data Store Using TADOConnection 1478
Connecting to a Database Server 1509
Connecting to a Database using the dbExpress Driver Framework 76
Connecting to a Server 1424
Connecting to ADO Data Stores 1477
Connecting to an Application Server That Uses Multiple Data Modules 1526
Connecting to Another Dataset 1561
Connecting to Clients 2350
Connecting to databases 1495
Connecting to databases with TDatabase 1644
Connecting to Databases with TDatabase 62
Connecting to Databases: Overview 1506
Connecting to the Application Server 1517
Connecting to the Application Server using DataSnap Components 55
Connecting to the Database Server 1825
Connecting to the Server 1519
connection components
client applications 1517
implicit 1505
connections
ADO 1477
asynchronous 1483
timing out 1480
Considerations When Dynamically Resizing Forms and Controls 1946
Considerations When Not Dynamically Resizing 1947

Console Applications 1900
const 538
keywords, C++ 538
const_cast
typecasting, C++ 539
const_cast (typecast Operator) 539
constant expressions
lexical elements, C++ 670
Constant Expressions 670
constants
lexical elements, C++ 667
Constants 667
Constants and Internal Representation 670
Constants And Internal Representation 671
Constants Overview 662
constants, character
lexical elements, C++ 663
constants, enumeration
lexical elements, C++ 673
constants, floating-point
lexical elements, C++ 672
constants, integer
lexical elements, C++ 669, 674
constants, internal representation
lexical elements, C++ 671, 676
constants, string
lexical elements, C++ 677
constants, wide-character
lexical elements, C++ 666
constexpr 576
keywords, C++ 576
Constraining Data Values 1713
constraints
client datasets 1716
custom 1855
data integrity 1814
fields 1863
server 1863
constructor
declaring 1318
Constructor Defaults 419
constructors
C++ language specifics 415, 419
overriding 1365
Constructors 416, 419
Constructors And Destructors 415
Constructors In Exception Handling (C++) 2017
constructors, copy
C++ language specifics 421
constructors, defaults
C++ language specifics 419
constructors, order of calling
C++ language specifics 420
constructors, overloading
C++ language specifics 421
continue 540
keywords, C++ 540
control placement 2196
Controlling Access 1371
Controlling an Automation Server Using a Dispatch Interface 1424
Controlling an Automation Server Using a Dual Interface 1424
Controlling an Imported Object 1422
Controlling and Masking User Input 1878
Controlling Application Behavior 2007
Controlling Connections 1497
Controlling Grid Drawing 1753
Controlling How Child Controls Are Docked 2157
Controlling How Child Controls Are Undocked 2158
Controlling How Child Controls Respond to Drag-and-dock Operations 2158
Controlling Read/Write Access to Local Tables 1670
Controlling Read/Write Access to Tables 1587
Controlling Server Login 1507
Controlling Timeouts 1480
Controlling What Information Is Included in Data Packets 1812
Controlling When Forms Reside in Memory 1975
controls
ancestor classes 1316

data-aware 1333, 1777
graphic controls 1318
subclassing Windows controls 1321
up-down controls 2052
windowed 1316
conversion utilities
measurements 2092, 2105
conversions
string to PChar 2131
string types 2127
Converting an Unnamed Thread to a Named Thread 2239
Converting Field Values 1874
Converting HTML-transparent Tags 2269
Converting Measurements 2092
Converting To Strings With # 695
Converting XML Documents into Data Packets 1841
cool bars 2040
adding 1968
setting appearance 2003
Cool Bars (VCL Only) 2040
Coordinating threads 2225
Coordinating Threads 2227
Copying a Complete String List 2135
Copying a Complete String List (VCL) 107
Copying a File 2123
Copying an Item 1915
Copying and Clearing Custom Variants 2093
Copying Data from Another Dataset 1714
Copying Data from One Stream to Another 2094
Copying Data From One Stream To Another 106
Copying Graphics to the Clipboard 2186
cos, cosl 943
cosh, coshl 943
Counting the Strings in a List 2135
cpp32 preprocessor directives 710
CPP32.EXE, the C Compiler Preprocessor 167
cprintf 744
cputs 745
creating
data browsing control 1330
data editing controls 1330
graphic component 1220
Creating a Batch Move Component 1679
Creating a Class to Enable the Custom Variant Type 2094
Creating a Component with the Component Wizard 1321
Creating a Custom Constraint 1855
Creating a Customized Grid 1754
Creating a Data Browsing Control 1330
Creating a Data Editing Control 1330
Creating a Form Instance Using a Local Variable 2010
Creating a graphic component 1218
Creating a Graphic Component 1220
Creating a Grid That Contains Other Data-aware Controls 1755
Creating a Group of Speed Buttons 1979
Creating a New Component 1317
Creating a New Dataset 1714
Creating a New Property Page 1416
Creating a New String List 2135
Creating a New Type Library 1460
Creating a New VCL Component 97
Creating a New VCL for the Web Application 2255
Creating a Package 2213
Creating a Property Page for an ActiveX Control 1415
Creating a Simple Conversion Family and Adding Units 2095
Creating a Unit File 1317
Creating a VCL Form Instance Using a Local Variable 110
Creating a VCL Forms ActiveX Active Form 140
Creating a VCL Forms ActiveX Button 139
Creating Actions in a VCL Forms Application 100
Creating Active Server Pages: Overview 1400
Creating Additional Sessions 1652
Creating an Active Form for the Client Application 1526
Creating an Active Server Object 1401
Creating an Active Server Page 1400
Creating an Application that Uses Ribbon Controls 143
Creating an Index with IndexFieldNames 1581
Creating an OnUpdateRecord Event Handler 1693
Creating and Deleting Tables 1587

Creating and Editing Packages 2213
Creating and Editing Standard Data Modules 1911
Creating and Managing Menus 1975
Creating and Managing Off-screen Bitmaps 1378
Creating and Modifying Server Metadata 1826
Creating and Registering the Component (Dialog Box) 1360
Creating and registering the component (Grid) 1263
Creating and registering the data-aware component 1331
Creating and Registering the Graphic Component 1220
Creating and Registering the Modified Component 1363
Creating and Using Component Templates 1975
Creating and Using Decision Graphs 1796
Creating and Using Decision Grids 1803
Creating Applications 1883
Creating Array Properties 1253
Creating ASPs for In-process or Out-of-process Servers 1405
Creating Attribute Sets for Field Components 1870
Creating Business Rules in a Data Module 1911
Creating Clients for Servers That Do Not Have a Type Library 1426
Creating COM clients 1418
Creating COM Clients 1419
Creating Components 1315
Creating Custom Exception Classes for Web Services 2305
Creating Customizable Toolbars and Menus 1976
Creating Decision Datasets with the Decision Query Editor 1802
Creating Decision Datasets with TQuery or TTable 1787
Creating Decision Graphs 1802
Creating Decision Grids 1788
Creating DLLs Containing VCL Components (C++) 1880
Creating Drawing Spaces 2128
Creating Dynamic Menus 1977
Creating events 1231
Creating Events: Overview 1233
Creating Filters 1590
Creating Forms and Projects 1279
Creating Forms Dynamically 1980
Creating Frames 1977
Creating Graphic Controls 1318
Creating Groups of Tool Buttons 1981
Creating Heterogenous Queries 1676
Creating HTTP response messages 2282
Creating HTTP Response Messages 2283
Creating international applications 1920
Creating International Applications: Overview 1923
Creating Internet Applications: Overview 2251
Creating Internet server applications 2243
Creating Master/detail Relationships 1590
Creating methods 1241
Creating Methods to Store and Load Property Values 1256
Creating Methods: Overview 1243
Creating Modeless Forms Such as Windows 1981
Creating Most Recently Used Lists 1977
Creating multi-tiered applications 1510
Creating Multi-tiered Applications: Overview 1518
Creating New Message Handlers 1304
Creating Nonvisual Components 1318
Creating Original Controls 1316
Creating packages and DLLs 1880
Creating Packages and DLLs 1882
Creating Persistent Columns 1756
Creating Persistent Fields 1874
Creating Projects 1936
Creating properties 1245
Creating Properties for Interfaces 1253
Creating Properties for Subcomponents 1254
Creating Properties: Overview 1249
Creating reports with Rave Reports 1551
Creating Resource DLLs 1931
Creating simple COM servers 1427
Creating Simple COM Servers: Overview 1430
Creating SQL Statements for Update Components 1696
Creating Strings 109
Creating Submenus 1978
Creating the Application Server 1527
Creating the Client Application 1528
Creating the Component Interface 1361
Creating the Entries 1346
Creating the Help File 1347

Creating Toolbars and Menus 1978
Creating Web Broker Applications 1885
Creating Web Pages with an InternetExpress Page Producer 1529
Creating Web server applications 1884
Creating Web Server Applications 1886
Creating Web Server Applications Using IntraWeb 2259
Creating Web Server Applications Using WebSnap 2315
Creating Web Server Applications with Web Broker 2287
Creating Web server applications with WebSnap 2311
Creating Web Server Applications with WebSnap 2313
Creating Web Services Applications 1885
Creating WebSnap Applications 1885
Creating, Instantiating, and Destroying Objects 2066
creatnew 893
creattemp 894
critical sections 2227
crosstabs
    crosstabulated data 1804
cscanf 746
ctime, _wctime 1201
ctype.h 766
custom interfaces
    interfaces 1439
Custom Interfaces 1439
Custom Packages 2214
custom variants
    binary operations 2100
    comparison operations 2102
    copying 2093
    defining 2094
    enabling 2116
    loading and saving 2104
    properties 2108
    typecasts 2097
    unary operations 2103
    utilities 2117
customizing a grid 1264
Customizing a grid 1258
Customizing a Grid: Overview 1264
Customizing Decision Graph Series 1799
Customizing Decision Graphs 1801
Customizing Drag and Drop with a Drag Object 2158
Customizing the ActiveX Control's Interface 1411
Customizing the IDE Help System 1891
Customizing the InternetExpress Page Producer Template 1529
Cutting Graphics to the Clipboard 2186
Cutting, Copying, and Pasting Text 2159
CW_DEFAULT #define 856
cwait 985
-D compile option 690

D

data
    analyzing 1567
    displaying 1858
data dictionary
    field attributes 1680
Data Dictionary
    field attributes 1870
Data Explorer
    executing SQL 74
    modifying connections 75
data formats
    assigning 1854
data grids
    customizing 1754
    default columns 1765
    drawing 1753
    event handling 1761
    runtime options DBGrid component 1764
data links
    adding to components 1327
data modules 1913
    accessing from a form 1910
    business rules 1911
    creating and editing 1911
    naming 1912
    overview 1912

placing components 1913
remote 1910
data packets
converting to XML documents 1845
field attributes 1812
optional parameters 1808
persistent fields 1819
provider options 1818
data sources
associating with datasets 1747
disabling and enabling 1774
events 1776
data-aware
controls 1778
data-aware controls
associating with datasets 1746
displaying data 1776
editing 1773, 1774
fields 1858
database
connections 1497
dbExpress 26, 27
Database
adding a new connection 73
browsing objects 52, 74
executing SQL 74
modifying connections 75
database applications 1919
BDE-based 1643
deploying 1948
multi-tiered applications 1919
reports 1568
Database architecture 1557
Database Architecture 1562
database connections
disconnecting 1656
dropping 1661
managing 1665
opening 1653
database navigator
help hints 1759
multiple datasets 1766
Database Procedures 50
Database Security 1569
database servers
connecting 1509, 1646
disconnecting 1497
databases
associating with sessions 1645
cached updates 1687
changing data 1629
client datasets 98
connecting 1646, 1825
connection components 1506
considerations 1572
data sources 98
database applications 1566
DataSet providers 98
datasets 1505
locating 1654
metadata 1501
security 1507, 1569
sessions 1647
transactions 1570
DataRequest method
client datasets 1712
dataset fields 1867
datasets
associating with databases 1668
BDE-enabled 1675
cached updates 1695
Dataset component 1632
HTML representation 2266, 2267
opening 1604
queries 1613, 1618
resolving 1810
states 1633
stored procedures 1618

tables 1620
types 1619, 1635
unidirectional 1836
updating 1337
DataSets
decision support 91
DataSnap
multi-tier database support 1952
dBASE index
specifying 1672
DBCtrlGrid component 1755
dbExpress
client DataSets 103
DataSet providers 103
debugging 1827
dbExpress 4 Feature Overview 22
dbExpress Components 16
dbExpress database applications
deploying 1941, 1942
dbExpress Framework 26
dbExpress Framework Compatibility 27
dbGo Overview 14
DBGrid component
DBGridColumns component 1766
DCC32
compiling C++ 169
DCC32.EXE 169
DCC32.EXE, the Delphi Command Line Compiler 169
DCOM
advantages 1546
debugging
Web Application Debugger 41
Debugging a WebSnap Application using the Web Application Debugger 150
Debugging a Wizard 1284
Debugging Applications 1937
Debugging C++ Applications with CodeGuard Error Reporting 3
Debugging dbExpress Applications 1827
Debugging dbExpress Applications using TSQLMonitor 55
Debugging Multi-threaded Applications 2241
Debugging server applications 2244
Debugging Server Applications 2245
Debugging Service Applications 1903
Debugging Web Applications That Are DLLs 2244
Deciding Which Runtime Packages to Use 2214
Decision Cube editor 1796
design time information 1794
dimension settings 1794
memory control 1793
Decision Cube Properties and Events 1788
decision cubes
activating dimensions 1792
binning 1793
DecisionCube component 1797
setting restrictions 1793
decision datasets
SQL statements 1787
decision graphs
creating 1802
customizing 1801
DecisionGraph component 1796
display options 1796
setting data series 1799
templates 1792
types 1786
Decision Graphs at Runtime 1788
Decision Grid Properties 1798
decision grids
creating 1788
drilling 1790
expanding and collapsing dimensions 1790
pivoting 1792
properties 1798
runtime behavior 1788
Decision Grids at Runtime 1788
Decision Pivot Properties 1791
decision pivots
DecisionPivot component 1798
properties 1791

runtime behaviors 1789
Decision Pivots: Runtime Behavior 1789
decision queries
specifying 1802
decision sources
pivot states 1803
decision support
decision cubes 91
decision graphs 91
decision grids 91
decision pivots 91
decision sources 91
decision support components 1790
components 1800
getting data 1799
memory management 1800
runtime behavior 1789
Decision Support Components and Memory Control 1800
Decision Support Components at Runtime 1789
DecisionGrid component
decision grids 1803
Declaration and Declarators 612
Declaration Syntax 600
declarations
language structure, C++ 603
Declarations 593
Declarations And Definitions 625
Declarations And Prototypes 633
declarations, external
language structure, C++ 606
declarations, incomplete
language structure, C++ 653
declarations, namespaces
namespaces, C++ 427
Declaring a Message Identifier 1304
Declaring a Message-structure Type 1305
Declaring A namespace 427
Declaring a New Component Class 1372
Declaring A New Constructor (C++) 1318
Declaring a New Message-handling Method 1305
Declaring and Initializing Strings 2123
declaring component types
deriving classes 1372
declaring methods
methods 1243
Declaring Methods 1243
Declaring the Access Properties 1221
Declaring the Access Properties for a Data-aware Control 1331
Declaring the Class Field 1332
Declaring the Class Fields 1222
Declaring the Event 1240
Declaring the Property 1222
Declaring the Property Type 1223
Declaring the Register Procedure 1347
decltype 545
keywords, C++ 545
default 545
keywords, C++ 545
Default Exception Handling in VCL 2029
default property values
properties 1253
Default Property Values 1253
Default Runtime Libraries 628
defined 689
Defined 689
defining
namespaces, C++ 428
Defining a Calculated Field 1872
Defining a COM Object's Interface 1435
Defining a Data Field 1855
Defining a Lookup Field 1868
Defining a Lookup List Column 1757
Defining A namespace 428
Defining an Aggregate Field 1856
Defining And Undefining Macros 689
Defining and Using SOAP Headers 2303
Defining Custom Variants 2096
Defining New Classes 1370

Defining New Classes (Delphi) 2072
Defining New Persistent Fields 1856
Defining Properties 1251
Defining Protected Blocks 2017
Defining the Component Interface 1361
Defining the Component Writer's Interface 1373
Defining the Design-time Interface 1372
Defining the Handler Type 1239
Defining the Runtime Interface 1373
Defining the Thread Object 126
Defining thread objects 2231
Defining Thread Objects 2233
Defining Transformations 1843
Defining Your Own Events 1238
Defining Your Own Messages 1306
Defining Your Own VCL Exceptions 2029
definitions
    language structure, C++ 602
Definitions 634
delayimp.h 787
delete 546
    C++ language specifics 546
Deleting a File 2124
Deleting a String from a List 2137
Deleting and Switching Indexes 1715
Deleting Event Handlers 2145
Deleting Persistent Columns 1757
Deleting Persistent Field Components 1857
Deleting Records 1591
Deleting Selected Text 2160
Deleting Strings 112
Deleting Toolbar Buttons 1284
delline 747
Delphi programming fundamentals 1935
delta packets
    editing 1813
Demo Programs: Actions, Action Lists, Menus, and Toolbars 1981
Deploying ActiveX Controls 1945
Deploying applications 1938
Deploying Applications 1937
Deploying Applications: Overview 1947
Deploying BDE Applications 1941
Deploying Database Applications 1948
Deploying dbExpress Database Applications 1942
Deploying General Applications 1949
Deploying Multi-tiered Database Applications (DataSnap) 1952
Deploying On Apache Servers 1943
Deploying Packages 2215
Deploying Type Libraries 1467
Deploying Web Applications 1949
Deriving a Property-editor Class 1348
deriving classes
    property editors 1348
Deriving New Classes 1370
Deriving the Component 1318
Describing sockets 2335
Describing Sockets 2336
Describing the Content 2284
Describing the Host 2336
Describing the Server Application 2284
Designing a COM Object 1430
Designing an ActiveX Control 1409
Designing Applications 1935
Designing database applications 1556
Designing Database Applications: Overview 1566
Designing the user interface 1567
Designing the User Interface 1567
Designing Toolbars and Cool Bars 1982
Design-time Packages 2215
destructors
    C++ language specifics 423
Destructors 422, 423
destructors, directives and
    C++ language specifics 423, 425, 426
destructors, invoking
    C++ language specifics 424
Determining Dataset States 1633
Determining the Order of Merged Menu Items: GroupIndex Property 1982

Determining the Source of Data 1812
Determining What to Draw 1223
Determining What to Store 1255
Determining When Action Items Fire 2279
Developing a Windows Application 89
Developing Applications with VCL Components 36
Developing COM Applications 29
Developing COM-based Applications 1381
Developing Database Applications 1469
Developing Database Applications for the Win32 Platform 13
Developing Dialog Boxes 1988
Developing Interoperable Applications 29
Developing Reports for Your Win32 Applications 35
Developing the application user interface 1955
Developing the Application User Interface: Overview 1983
Developing Web Applications with WebSnap 41
Developing Web Services with Win32 Applications 44
Developing Windows Applications 45
dialog
    Windows Forms 88
dialog box as component
    component writing 1362
dialog boxes
    common dialogs 1988
difftime 1202
dir.h 799
Direct Access 1251
direct.h 790
Directives for MAKE 186
dirent.h 792
disable, _disable, enable, _enable 825
Disabling and Enabling Data Display 1768
Disabling Menu Items 2160
Disconnecting from a Database Server 1497
Dispatch Interfaces 1438
Dispatcher Components 2319
Dispatching Action Items 2320
Dispatching Messages 1306
Dispatching Methods 1373
Dispatching Request Messages 2281
Dispatching requests and responses 2316
Dispatching Requests and Responses 2320
dispid
    keywords, C++ 525
Display Controls 2052
Displaying a Bitmap Image in a VCL Forms Application 114
Displaying a Full View Bitmap Image in a VCL Forms Application 116
Displaying ADT and Array Fields 1758
Displaying an Auto-created Form 1983
Displaying an Auto-Created VCL Form 113
Displaying and Coding Shared Events 2145
Displaying and Editing a Subset of Data Using Filters 1631
Displaying and Editing Data in List and Combo Boxes 1768
Displaying and Editing Data in Lookup List and Combo Boxes 1769
Displaying and Editing Fields in an Edit Box 1770
Displaying and Editing Graphics Fields in an Image Control 1771
Displaying and Editing Text in a Memo Control 1771
Displaying and Editing Text in a Rich Edit Memo Control 1772
displaying bitmap images 114, 116
displaying data
    disabling and enabling 1768
Displaying Data as Labels 1772
Displaying Field Component Values in Standard Controls 1858
Displaying Fly-over Help 1759
Displaying Keyword-based Help 1892
Displaying Multiple Records 1760
Displaying Tables of Contents 1893
Displaying, Converting, and Accessing Field Values 1858
distributed applications
    interfaces 2079
Distributing a Client Application as an ActiveX Control 1530
Distributing Database Applications 1919
div 1111
Divide error (C++) 309
division
    operators, C++ 593
DLL Locations 1947

DLLs
creating that contain VCL components 1880
dynamic link libraries, using 1883
linking 1883
locations 1947
do 546
keywords, C++ 546
docking
dockable child controls 2157, 2158, 2164
DOM
Document Object Model 2353
dos.h 818
dostounix 825
double 547
keywords, C++ 547, 555
double-byte character sets
two-byte character codes 1925
drag and drop 2168
drag-and-dock
docking 2158, 2163
drag-and-drop 2153, 2158, 2160, 2162
Draw Grids 2040
drawing
polygons 117
rectangles and ellipses 118
round rectangles 118
straight lines 119
Drawing a Polygon in a VCL Forms Application 117
Drawing a Rounded Rectangle in a VCL Forms Application 118
Drawing Lines 2187
Drawing Lines and Polylines 2187
drawing objects 2192
Drawing On a Graphic 2187
Drawing On the Bitmap 2188
Drawing Owner-draw Items 2161
Drawing Owner-drawn Items 2161
Drawing Polygons 2189
Drawing Polylines 2189
Drawing Rectangles and Ellipses 2189
Drawing Rectangles and Ellipses in a VCL Forms Application 118
Drawing Rounded Rectangles 2189
Drawing Shapes 2190
Drawing Shapes (Code) 2190
Drawing Straight Lines In a VCL Forms Application 119
Drawing the Component Image 1224
drawing tools 2193, 2206
Drilling Down for Detail in Decision Grids 1790
drivers
dbExpress 1834
Dropping Inactive Database Connections 1661
Dropping Items 2162
Dropping or Changing a Server Connection 1531
dual interface
Automation controller 1424
interfaces 1438
Dual Interfaces 1438
dup 895
dup2 896
duration
language structure, C++ 594
Duration 594
Dynamic Field Components 1876
dynamic functions
C++ language specifics 447
Dynamic Functions 447
Dynamic Methods 1373
Dynamic Switching of Resource DLLs 1934
dynamic_cast
typecasting, C++ 547
dynamic_cast (typecast Operator) 547
Dynamically Creating a VCL Modal Form 120
Dynamically Creating a VCL Modeless Form 121
Dynamic-link Libraries 629

E

E2000: 286/287 instructions not enabled (C++) 286
E2001: Constructors and destructors not allowed in __automated section (C++) 289

E2002: Only __fastcall functions allowed in __automated section (C++) 289

E2003: Data member definition not allowed in __automated section (C++) 289

E2004: Only read or write clause allowed in property declaration in __automated section (C++) 289

E2005: Redeclaration of property not allowed in __automated section (C++) 290

E2006: CodeGuarded programs must use the large memory model and be targeted for Windows (C++) 342

E2007: Dispid only allowed in __automated sections (C++) 309

E2008: Published property access functions must use __fastcall calling convention (C++) 257

E2009: Attempt to grant or reduce access to 'identifier' (C++) 286

E2010: Cannot add or subtract relocatable symbols (C++) 287

E2011: Illegal to take address of bit field (C++) 286

E2012: Cannot take address of 'main' (C++) 344

E2013: 'function1' cannot be distinguished from 'function2' (C++) 287

E2014: Member is ambiguous: 'member1' and 'member2' (C++) 287

E2015: Ambiguity between 'function1' and 'function2' (C++) 287

E2016: Ambiguous override of virtual base member 'base_function': 'derived_function' (C++) 344

E2017: Ambiguous member name 'name' (C++) 287

E2018: Cannot catch 'identifier' -- ambiguous base class 'identifier' (C++) 380

E2019: 'identifier' cannot be declared in an anonymous union (C++) 288

E2020: Global anonymous union not static (C++) 288

E2021: Array must have at least one element (C++) 344

E2022: Array size too large (C++) 288

E2023: Array of references is not allowed (C++) 344

E2024: Cannot modify a const object (C++) 288

E2025: Assignment to 'this' not allowed, use X::operator new instead (C++) 288

E2026: Assembler statement too long (C++) 288

E2027: Must take address of a memory location (C++) 290

E2028: operator -> must return a pointer or a class (C++) 290

E2029: 'identifier' must be a previously defined class or struct (C++) 290

E2030: Misplaced break (C++) 290

E2031: Cannot cast from 'type1' to 'type2' (C++) 290

E2032: Illegal use of closure pointer (C++) 344

E2033: Misplaced continue (C++) 291

E2034: Cannot convert 'type1' to 'type2' (C++) 291

E2035: Conversions of class to itself or base class not allowed (C++) 304

E2036: Conversion operator cannot have a return type specification (C++) 292

E2037: The constructor 'constructor' is not allowed (C++) 292

E2038: Cannot declare or define 'identifier' here: wrong namespace (C++) 304

E2039: Misplaced decimal point (C++) 292

E2040: Declaration terminated incorrectly (C++) 345

E2041: Incorrect use of default (C++) 292

E2042: Declare operator delete (void*) or (void*, size_t) (C++) 293

E2044: operator delete must return void (C++) 293

E2045: Destructor name must match the class name (C++) 293

E2046: Bad file name format in include directive OR Bad file name format in line directive (C++) 294

E2047: Bad 'directive' directive syntax (C++) 345

E2048: Unknown preprocessor directive: 'identifier' (C++) 293

E2049: Class type 'type' cannot be marked as __declspec(delphireturn) (C++) 345

E2050: __declspec(delphireturn) class 'class' must have exactly one data member (C++) 258

E2051: Invalid use of dot (C++) 294

E2052: Dynamic function 'function' conflicts with base class 'class' (C++) 345

E2053: Misplaced elif directive (C++) 294

E2054: Misplaced else (C++) 294

E2055: Misplaced else directive (C++) 294

E2056: Misplaced endif directive (C++) 295

E2057: Exception specification not allowed here (C++) 345

E2058: Exception handling variable may not be used here (C++) 346

E2059: Unknown language, must be C or C++ (C++) 295

E2060: Illegal use of floating point (C++) 295

E2061: Friends must be functions or classes (C++) 295

E2062: Invalid indirection (C++) 295

E2063: Illegal initialization (C++) 296

E2064: Cannot initialize 'type1' with 'type2' (C++) 296

E2065: Using namespace symbol 'symbol' conflicts with intrinsic of the same name (C++) 346

E2066: Information not available (C++) 261

E2066: Invalid MOM inheritance (C++) 256

E2067: 'main' must have a return type of int (C++) 346

E2068: 'identifier' is not a non-static data member and can't be initialized here (C++) 296

E2069: Illegal use of member pointer (C++) 296

E2070: Invalid use of namespace 'identifier' (C++) 315

E2071: operator new must have an initial parameter of type size_t (C++) 297

E2072: Operator new[] must return an object of type void (C++) 297

E2073: Nothing allowed after pragma option pop (C++) 346

E2074: Value after -g or -j should be between 0 and 255 inclusive (C++) 259

E2075: Incorrect 'type' option: option (C++) 297

E2076: Overloadable operator expected (C++) 297

E2077: 'operator' must be declared with one or no parameters (C++) 298

E2078: 'operator' must be declared with one or two parameters (C++) 298

E2079: 'function' must be declared with no parameters (C++) 298

E2080: 'function' must be declared with one parameter (C++) 298

E2081: 'function' must be declared with two parameters (C++) 298

E2082: 'identifier' must be a member function or have a parameter of class type (C++) 298

E2083: Last parameter of 'operator' must have type 'int' (C++) 299

E2084: Parameter names are used only with a function body (C++) 299

E2085: Invalid pointer addition (C++) 299

E2086: Illegal pointer subtraction (C++) 299

E2087: Illegal use of pointer (C++) 299

E2088: Bad syntax for pure function definition (C++) 300

E2089: Identifier 'identifier' cannot have a type qualifier (C++) 300

E2090: Qualifier 'identifier' is not a class or namespace name (C++) 300

E2091: Functions cannot return arrays or functions (C++) 346

E2092: Storage class 'storage class' is not allowed here (C++) 300

E2093: Operator 'operator' not implemented in type 'type' for arguments of the same type (C++) 346

E2094: Operator 'operator' not implemented in type 'type' for arguments of type 'type' (C++) 346

E2095: String literal not allowed in this context (C++) 343

E2096: Illegal structure operation (C++) 300

E2097: Explicit instantiation only allowed at file or namespace scope (C++) 347

E2098: Explicit specialization declarator "template<>" now required (C++) 347

E2099: Explicit specialization only allowed at file or namespace scope (C++) 347

E2100: Invalid template declarator list (C++) 259

E2101: 'export' keyword must precede a template declaration (C++) 347

E2102: Cannot use template 'template' without specifying specialization parameters (C++) 260

E2103: Explicit instantiation must be used with a template class or function (C++) 347

E2104: Invalid use of template keyword (C++) 300

E2105: 'template' qualifier must specify a member template name (C++) 260

E2106: Explicit specialization must be used with a template class or function (C++) 347

E2107: Invalid use of template 'template' (C++) 260

E2108: Improper use of typedef 'identifier' (C++) 301

E2109: Not an allowed type (C++) 301

E2110: Incompatible type conversion (C++) 301

E2111: Type 'typename' may not be defined here (C++) 302

E2112: Unknown unit directive: 'directive' (C++) 347

E2113: Virtual function 'function1' conflicts with base class 'base' (C++) 301

E2114: Multiple base classes require explicit class names (C++) 301

E2115: Bit field too large (C++) 301

E2116: Bit fields must contain at least one bit (C++) 301

E2117: Bit fields must be signed or unsigned int (C++) 363

E2118: Bit fields must have integral type (C++) 348

E2119: User break (C++) 302

E2120: Cannot call 'main' from within the program (C++) 348

E2121: Function call missing) (C++) 302

E2122: Function call terminated by unhandled exception 'value' at address 'addr' (C++) 257

E2123: Class 'class' may not contain pure functions (C++) 302

E2124: Invalid function call (C++) 343

E2125: Compiler could not generate copy constructor for class

'class' OR Compiler could not generate default constructor for class 'class' OR Compiler could not generate operator = for class 'class' (C++) 348

E2126: Case bypasses initialization of a local variable (C++) 302

E2127: Case statement missing : (C++) 302

E2128: Case outside of switch (C++) 302

E2129: Character constant too long (or empty) (C++) 303

E2130: Circular property definition (C++) 348

E2131: Objects of type 'type' cannot be initialized with { } (C++) 348

E2132: Templates and overloaded operators cannot have C linkage (C++) 339

E2133: Unable to execute command 'command' (C++) 303

E2134: Compound statement missing closing brace (C++) 303

E2135: Constructor/Destructor cannot be declared 'const' or 'volatile' (C++) 303

E2136: Constructor cannot have a return type specification (C++) 303

E2137: Destructor for 'class' required in conditional expression (C++) 303

E2138: Conflicting type modifiers (C++) 303

E2139: Declaration missing ; (C++) 304

E2140: Declaration is not allowed here (C++) 304

E2141: Declaration syntax error (C++) 304

E2142: Base class 'class' contains dynamically dispatchable functions (C++) 304

E2143: Matching base class function 'function' has different dispatch number (C++) 305

E2144: Matching base class function 'function' is not dynamic (C++) 305

E2145: Functions 'function1' and 'function2' both use the same dispatch number (C++) 305

E2146: Need an identifier to declare (C++) 305

E2147: 'identifier' cannot start a parameter declaration (C++) 305

E2148: Default argument value redeclared for parameter 'parameter' (C++) 348

E2149: Default argument value redeclared (C++) 349

E2150: Type mismatch in default argument value (C++) 305

E2151: Type mismatch in default value for parameter 'parameter' (C++) 349

E2152: Default expression may not use local variables (C++) 306

E2153: Define directive needs an identifier (C++) 306

E2154: Cannot define 'identifier' using a namespace alias (C++) 304

E2155: Too many default cases (C++) 306

E2156: Default outside of switch (C++) 306

E2157: Deleting an object requires exactly one conversion to pointer operator (C++) 349

E2158: Operand of 'delete' must be non-const pointer (C++) 306

E2159: Trying to derive a far class from the huge base 'base' (C++) 306

E2160: Trying to derive a far class from the near base 'base' (C++) 306

E2161: Trying to derive a huge class from the far base 'base' (C++) 307

E2162: Trying to derive a huge class from the near base 'base' (C++) 307

E2163: Trying to derive a near class from the far base 'base' (C++) 307

E2164: Trying to derive a near class from the huge base 'base' (C++) 307

E2165: Destructor cannot have a return type specification (C++) 307

E2166: Destructor for 'class' is not accessible (C++) 307

E2167: 'function' was previously declared with the language 'language' (C++) 307

E2168: Division by zero (C++) 308

E2169: 'identifier' specifies multiple or duplicate access (C++) 308

E2170: Base class 'class' is included more than once (C++) 308

E2171: Body has already been defined for function 'function' (C++) 308

E2172: Duplicate case (C++) 308

E2173: Duplicate handler for 'type1', already had 'type2' (C++) 349

E2174: The name handler must be last (C++) 349

E2175: Too many storage classes in declaration (C++) 308

E2176: Too many types in declaration (C++) 309

E2177: Redeclaration of #pragma package with different arguments (C++) 350

E2178: VIRDEF name conflict for 'function' (C++) 350

E2179: virtual specified more than once (C++) 309

E2180: Dispid number already used by identifier (C++) 350

E2181: Cannot override a 'dynamic/virtual' with a 'dynamic/virtual' function (C++) 350

E2182: Illegal parameter to __emit__ (C++) 309

E2183: File must contain at least one external declaration (C++) 309

E2184: Enum syntax error (C++) 310

E2185: The value for 'identifier' is not within the range of 'type-name' (C++) 310

E2186: Unexpected end of file in comment started on line 'number' (C++) 310

E2187: Unexpected end of file in conditional started on line 'number' (C++) 310

E2188: Expression syntax (C++) 310

E2189: extern variable cannot be initialized (C++) 311

E2190: Unexpected closing brace (C++) 311

E2191: '__far16' may only be used with '__pascal' or '__cdecl' (C++) 261

E2192: Too few parameters in call (C++) 311

E2193: Too few parameters in call to 'function' (C++) 311

E2194: Could not find file 'filename' (C++) 311

E2195: Cannot evaluate function call (C++) 312

E2196: Cannot take address of member function 'function' (C++) 284

E2197: File name too long (C++) 311

E2198: Not a valid expression format type (C++) 312

E2199: Template friend function 'function' must be previously declared (C++) 262

E2200: Functions may not be part of a struct or union (C++) 312

E2201: Too much global data defined in file (C++) 313

E2202: Goto into an exception handler is not allowed (C++) 350

E2203: Goto bypasses initialization of a local variable (C++) 313

E2204: Group overflowed maximum size: 'name' (C++) 313

E2205: Illegal type type in __automated section (C++) 350

E2206: Illegal character 'character' (0x'value') (C++) 313

E2207: Implicit conversion of 'type1' to 'type2' not allowed (C++) 313

E2208: Cannot access an inactive scope (C++) 313

E2209: Unable to open include file 'filename' (C++) 314

E2210: Reference member 'member' is not initialized (C++) 314

E2211: Inline assembly not allowed in inline and template functions (C++) 314

E2212: Function defined inline after use as extern (C++) 314

E2213: Invalid 'expression' in scope override (C++) 343

E2214: Cannot have a 'non-inline function/static data' in a local class (C++) 315

E2215: Linkage specification not allowed (C++) 315

E2216: Unable to create turboc.$ln (C++) 315

E2217: Local data exceeds segment size limit (C++) 316

E2218: Templates can only be declared at namespace or class scope (C++) 315

E2219: Wrong number of arguments in call of macro 'macro' (C++) 316

E2220: Invalid macro argument separator (C++) 316

E2221: Macro argument syntax error (C++) 316

E2222: Macro expansion too long (C++) 316

E2223: Too many decimal points (C++) 316

E2224: Too many exponents (C++) 316

E2225: Too many initializers (C++) 316

E2226: Extra parameter in call (C++) 317

E2227: Extra parameter in call to function (C++) 317

E2228: Too many error or warning messages (C++) 317

E2229: Member 'member' has the same name as its class (C++) 317

E2230: In-line data member initialization requires an integral constant expression (C++) 264

E2231: Member 'member' cannot be used without an object (C++) 318

E2232: Constant/Reference member 'member' in class without constructors (C++) 317

E2233: Cannot initialize a class member here (C++) 317

E2234: Memory reference expected (C++) 317

E2235: Member function must be called or its address taken (C++) 318

E2236: Missing 'identifier' in scope override (C++) 343

E2238: Multiple declaration for 'identifier' (C++) 318

E2239: 'identifier' must be a member function (C++) 319

E2240: Conversion of near pointer not allowed (C++) 319

E2241: VCL style classes need virtual destructors (C++) 264

E2242: Specifier requires Delphi style class type (C++) 351

E2243: Array allocated using 'new' may not have an initializer (C++) 319

E2244: 'new' and 'delete' not supported (C++) 319

E2245: Cannot allocate a reference (C++) 319

E2246: x is not abstract public single inheritance class hierarchy with no data (C++) 264

E2247: 'member' is not accessible (C++) 351

E2248: Cannot find default constructor to initialize array element of type 'class' (C++) 351

E2249: = expected (C++) 264

E2250: No base class to initialize (C++) 320

E2251: Cannot find default constructor to initialize base class 'class' (C++) 352

E2252: 'catch' expected (C++) 352

E2253: Calling convention must be attributed to the function type, not the closure (C++) 352

E2254: : expected after private/protected/private (C++) 320

E2255: Use :: to take the address of a member function (C++) 320

E2256: No : following the ? (C++) 320

E2257: , expected (C++) 320

E2258: Declaration was expected (C++) 320

E2259: Default value missing (C++) 321

E2260: Default value missing following parameter 'parameter' (C++) 321

E2261: Use of dispid with a property requires a getter or setter (C++) 353

E2262: '__except' or '__finally' expected following '__try' (C++) 353

E2263: Exception handling not enabled (C++) 321

E2264: Expression expected (C++) 321

E2265: No file name ending (C++) 321

E2266: No file names given (C++) 321

E2267: First base must be VCL class (C++) 264

E2268: Call to undefined function 'function' (C++) 360

E2269: The function 'function' is not available (C++) 342

E2270: > expected (C++) 353

E2271: Goto statement missing label (C++) 321

E2272: Identifier expected (C++) 321

E2273: 'main' cannot be declared as static or inline (C++) 353

E2274: < expected (C++) 322

E2275: Opening brace expected (C++) 322

E2276: (expected (C++) 322

E2277: Lvalue required (C++) 322

E2278: Multiple base classes not supported for Delphi classes (C++) 322

E2279: Cannot find default constructor to initialize member 'identifier' (C++) 323

E2280: Member identifier expected (C++) 322

E2281: Identifier1 requires definition of Identifier2 as a pointer type (C++) 353

E2282: Namespace name expected (C++) 328

E2283: Use . or -> to call 'function' (C++) 323

E2284: Use . or -> to call 'member', or & to take its address (C++) 323

E2285: Could not find a match for 'argument(s)' (C++) 323

E2286: Overloaded function resolution not supported (C++) 324

E2287: Parameter 'number' missing name (C++) 324

E2288: Pointer to structure required on left side of -> or ->* (C++) 324

E2289: __published or __automated sections only supported for Delphi classes (C++) 353

E2290: 'code' missing] (C++) 324

E2291: brace expected (C++) 324

E2292: Function should return a value (C++) 324

E2293:) expected (C++) 325

E2294: Structure required on left side of . or .* (C++) 325

E2295: Too many candidate template specializations from 'specifier' (C++) 266

E2296: Templates not supported (C++) 325

E2297: 'this' can only be used within a member function (C++) 326

E2298: Cannot generate 'function' from template function 'template' (C++) 354

E2299: Cannot generate template specialization from 'specifier' (C++) 267

E2300: Could not generate a specialization matching type for 'specifier' (C++) 267

E2301: Cannot use templates in closure arguments -- use a typedef (C++) 354

E2302: No type information (C++) 327

E2303: Type name expected (C++) 327

E2304: 'Constant/Reference' variable 'variable' must be initialized (C++) 327

E2305: Cannot find 'class::class' ('class'&) to copy a vector OR Cannot find 'class'::operator=('class'&) to copy a vector (C++) 327

E2306: Virtual base classes not supported for Delphi classes (C++) 328

E2307: Type 'type' is not a defined class with virtual functions (C++) 354

E2308: do statement must have while (C++) 328

E2309: Inline assembly not allowed (C++) 320

E2310: Only member functions may be 'const' or 'volatile' (C++) 323

E2311: Non-virtual function 'function' declared pure (C++) 323

E2312: 'constructor' is not an unambiguous base class of 'class' (C++) 325

E2313: Constant expression required (C++) 325

E2314: Call of nonfunction (C++) 325

E2315: 'Member' is not a member of 'class', because the type is not yet defined (C++) 354

E2316: 'identifier' is not a member of 'struct' (C++) 326

E2317: 'identifier' is not a parameter (C++) 326

E2318: 'type' is not a polymorphic class type (C++) 354

E2319: 'identifier' is not a public base class of 'classtype' (C++) 326

E2320: Expression of scalar type expected (C++) 326

E2321: Declaration does not specify a tag or an identifier (C++) 325

E2322: Incorrect number format (C++) 328

E2323: Illegal number suffix (C++) 355

E2324: Numeric constant too large (C++) 328

E2325: Illegal octal digit (C++) 329

E2326: Use __declspec(spec1[, spec2]) to combine multiple __declspecs (C++) 355

E2327: Operators may not have default argument values (C++) 329

E2328: Classes with properties cannot be copied by value (C++) 355

E2329: Invalid combination of opcode and operands (C++) 329

E2330: Operator must be declared as function (C++) 329

E2331: Number of allowable option contexts exceeded (C++) 355

E2332: Variable 'variable' has been optimized and is not available (C++) 355

E2333: Class member 'member' declared outside its class (C++) 329

E2334: Namespace member 'identifier' declared outside its namespace (C++) 328

E2335: Overloaded 'function name' ambiguous in this context (C++) 329

E2336: Pointer to overloaded function 'function' doesn't match 'type' (C++) 330

E2337: Only one of a set of overloaded functions can be "C" (C++) 330

E2338: Overlays only supported in medium, large, and huge memory models (C++) 330

E2339: Cannot overload 'main' (C++) 330

E2340: Type mismatch in parameter 'number' (C++) 330

E2341: Type mismatch in parameter 'number' in call to 'function' (C++) 331

E2342: Type mismatch in parameter 'parameter' (C++) 331

E2343: Type mismatch in parameter 'parameter' in call to 'function' (C++) 331

E2344: Earlier declaration of 'identifier' (C++) 311

E2345: Access can only be changed to public or protected (C++) 331

E2346: 'x' access specifier of property 'property' must be a member function (C++) 356

E2347: Parameter mismatch in access specifier 'specifier' of property 'property' (C++) 356

E2348: Storage specifier not allowed for array properties (C++) 356

E2349: Nonportable pointer conversion (C++) 331

E2350: Cannot define a pointer or reference to a reference (C++) 332

E2351: Static data members not allowed in __published or __automated sections (C++) 357

E2352: Cannot create instance of abstract class 'class' (C++) 332

E2353: Class 'classname' is abstract because of 'member = 0' (C++) 357

E2354: Two operands must evaluate to the same type (C++) 332

E2355: Recursive template function: 'x' instantiated 'y' (C++) 333

E2356: Type mismatch in redeclaration of 'identifier' (C++) 333

E2357: Reference initialized with 'type1', needs lvalue of type 'type2' (C++) 333

E2358: Reference member 'member' needs a temporary for initialization (C++) 333

E2359: Reference member 'member' initialized with a non-reference parameter (C++) 357

E2360: Invalid register combination (e.g. [BP+BX]) (C++) 334

E2361: 'specifier' has already been included (C++) 334

E2362: Repeat count needs an lvalue (C++) 334

E2363: Attempting to return a reference to local variable 'identifier' (C++) 334

E2364: Attempting to return a reference to a local object (C++) 334

E2365: Member pointer required on right side of .* or ->* (C++) 335

E2366: Can't inherit non-RTTI class from RTTI base OR E2367 Can't inherit RTTI class from non-RTTI base (C++) 335

E2368: RTTI not available for expression evaluation (C++) 335

E2369: Cannot use the result of a property assignment as an rvalue' (C++) 358

E2370: Simple type name expected (C++) 358

E2371: sizeof may not be applied to a bit field (C++) 335

E2372: sizeof may not be applied to a function (C++) 335

E2373: Bit field cannot be static (C++) 335

E2374: Function 'function' cannot be static (C++) 335

E2375: Assembler stack overflow (C++) 360

E2376: statement missing (C++) 336

E2377: statement missing) (C++) 336

E2378: do-while or for statement missing ; (C++) 336

E2379: Statement missing ; (C++) 336

E2380: Unterminated string or character constant (C++) 336

E2381: Structure size too large (C++) 336

E2382: Side effects are not allowed (C++) 336

E2383: Switch selection expression must be of integral type (C++) 336

E2384: Cannot call near class member function with a pointer of type 'type' (C++) 337

E2386: Cannot involve parameter 'parameter' in a complex partial specialization expression (C++) 268

E2387: Partial specializations may not specialize dependent non-type parameters ('parameter') (C++) 269

E2388: Argument list of specialization cannot be identical to the parameter list of primary template (C++) 269

E2389: Mismatch in kind of substitution argument and template parameter 'parameter' (C++) 269

E2390: Type mismatch in parameter 'number' in template class name 'template' (C++) 337

E2391: Type mismatch in parameter 'parameter' in template class name 'template' (C++) 337

E2392: Template instance 'template' is already instantiated (C++) 270

E2393: Cannot take the address of non-type, non-reference template parameter 'parameter' (C++) 270

E2394: Too few arguments passed to template 'template' (C++) 337

E2395: Too many arguments passed to template 'template' (C++) 338

E2396: Template argument must be a constant expression (C++) 338

E2397: Template argument cannot have static or local linkage (C++) 270

E2398: Template function argument 'argument' not used in argument types (C++) 358

E2399: Cannot reference template argument 'arg' in template class 'class' this way (C++) 270

E2400: Nontype template argument must be of scalar type (C++) 338

E2401: Invalid template argument list (C++) 338

E2402: Illegal base class type: formal type 'type' resolves to 'type' (C++) 271

E2403: Dependent call specifier yields non-function 'name' (C++) 271

E2404: Dependent type qualifier 'qualifier' has no member type named 'name' (C++) 271

E2405: Dependent template reference 'identifier' yields non-template symbol (C++) 271

E2406: Dependent type qualifier 'qualifier' is not a class or struct type (C++) 272

E2407: Dependent type qualifier 'qualifier' has no member symbol named 'name' (C++) 272

E2408: Default values may be specified only in primary class template declarations (C++) 272

E2409: Cannot find a valid specialization for 'specifier' (C++) 272

E2410: Missing template parameters for friend template 'template' (C++) 273

E2411: Declaration of member function default parameters after a specialization has already been expanded (C++) 273

E2412: Attempting to bind a member reference to a dependent type (C++) 273

E2413: Invalid template declaration (C++) 315

E2414: Destructors cannot be declared as template functions (C++) 273

E2415: Template functions may only have 'type-arguments' (C++) 338

E2416: Invalid template function declaration (C++) 274

E2417: Cannot specify template parameters in explicit specialization of 'specifier' (C++) 274

E2418: Maximum instantiation depth exceeded; check for recursion (C++) 274

E2419: Error while instantiating template 'template' (C++) 358

E2420: Explicit instantiation can only be used at global scope (C++) 274

E2421: Cannot use local type 'identifier' as template argument (C++) 304

E2422: Argument kind mismatch in redeclaration of template parameter 'parameter' (C++) 275

E2423: Explicit specialization or instantiation of non-existing template 'template' (C++) 275

E2424: Template class nesting too deep: 'class' (C++) 358

E2425: 'member' is not a valid template type member (C++) 338

E2426: Explicit specialization of 'specifier' requires 'template<>' declaration (C++) 275

E2427: 'main' cannot be a template function (C++) 276

E2428: Templates must be classes or functions (C++) 338

E2429: Not a valid partial specialization of 'specifier' (C++) 276

E2430: Number of template parameters does not match in redeclaration of 'specifier' (C++) 276

E2431: Non-type template parameters cannot be of floating point, class, or void type (C++) 277

E2432: 'template' qualifier must name a template class or function instance' (C++) 339

E2433: Specialization after first use of template (C++) 337

E2434: Template declaration missing template parameters ('template<...>') (C++) 277

E2435: Too many template parameter sets were specified (C++) 277

E2436: Default type for template template argument 'arg' does not name a primary template class (C++) 277

E2437: 'typename' should be followed by a qualified, dependent type name (C++) 278

E2438: Template template arguments must name a class (C++) 278

E2439: 'typename' is only allowed in template declarations (C++) 278

E2440: Cannot generate specialization from 'specifier' because that type is not yet defined (C++) 278

E2441: Instantiating 'specifier' (C++) 278

E2442: Two consecutive dots (C++) 339

E2443: Base class 'class' is initialized more than once (C++) 339

E2444: Member 'member' is initialized more than once (C++) 339

E2445: Variable 'identifier' is initialized more than once (C++) 339

E2446: Function definition cannot be a typedef'ed declaration (C++) 339

E2447: 'identifier' must be a previously defined enumeration tag (C++) 340

E2448: Undefined label 'identifier' (C++) 340

E2449: Size of 'identifier' is unknown or zero (C++) 340

E2450: Undefined structure 'structure' (C++) 340

E2451: Undefined symbol 'identifier' (C++) 340

E2452: Size of the type is unknown or zero (C++) 341

E2453: Size of the type 'identifier' is unknown or zero (C++) 340

E2454: union cannot be a base type (C++) 341

E2455: union cannot have a base type (C++) 341

E2456: Union member 'member' is of type class with 'constructor' (or destructor, or operator =) (C++) 341

E2457: Delphi style classes must be caught by reference (C++) 359

E2458: Delphi classes have to be derived from Delphi classes (C++) 359

E2459: Delphi style classes must be constructed using operator new (C++) 359

E2460: Delphi style classes require exception handling to be enabled (C++) 360

E2461: '%s' requires run-time initialization/finalization (C++) 341

E2462: 'virtual' can only be used with non-template member functions (C++) 279

E2463: 'base' is an indirect virtual base class of 'class' (C++) 360

E2464: 'virtual' can only be used with member functions (C++) 341

E2465: unions cannot have virtual member functions (C++) 342

E2466: void & is not a valid type (C++) 342

E2467: 'Void function' cannot return a value (C++) 342

E2468: Value of type void is not allowed (C++) 342

E2469: Cannot use tiny or huge memory model with Windows (C++) 342

E2470: Need to include header <typeinfo> to use typeid (C++) 279

E2471: pragma checkoption failed: options are not as expected (C++) 261

E2472: Cannot declare a member function via instantiation (C++) 265

E2473: Invalid explicit specialization of 'specifier' (C++) 274

E2474: 'function' cannot be declared as static or inline (C++) 265

E2475: 'function' cannot be a template function (C++) 266

E2476: Cannot overload 'function' (C++) 356

E2477: Too few template parameters were declared for template 'template' (C++) 276

E2478: Too many template parameters were declared for template 'template' (C++) 277

E2479: Cannot have both a template class and function named 'name' (C++) 275

E2480: Cannot involve template parameters in complex partial specialization arguments (C++) 269

E2481: Unexpected string constant (C++) 268

E2482: String constant expected (C++) 268

E2483: Array dimension 'specifier' could not be determined (C++) 258

E2484: The name of template class 'class' cannot be overloaded (C++) 275

E2485: Cannot use address of array element as non-type

template argument (C++) 271

E2486: Cannot use address of class member as non-type template argument (C++) 273

E2487: Cannot specify default function arguments for explicit specializations (C++) 276

E2488: Maximum token reply depth exceeded; check for recursion (C++) 263

E2489: Maximum option context replay depth exceeded; check for recursion (C++) 263

E2490: Specialization within template classes not yet implemented (C++) 274

E2491: Maximum VIRDEF count exceeded; check for recursion (C++) 263

E2492: Properties may only be assigned using a simple statement, e.g. \"prop = value;\" (C++) 259

E2493: Invalid GUID string (C++) 263

E2494: Unrecognized __declspec modifier (C++) 262

E2495: Redefinition of uuid is not identical (C++) 268

E2496: Invalid call to uuidof(struct type|variable) (C++) 263

E2497: No GUID associated with type:'type' (C++) 267

E2498: Need previously defined struct GUID (C++) 266

E2499: Invalid __declspec(uuid(GuidString)) format (C++) 263

E2500: __declspec(selectany) is only for initialized and externally visible variables (C++) 268

E2501: Unable to open import file 'filename' (C++) 262

E2502: Error resolving #import: problem (C++) 262

E2503: Missing or incorrect version of TypeLibImport.dll (C++) 279

E2504: 'dynamic' can only be used with non-template member functions (C++) 261

E2505: Explicit instantiation requires an elaborated type specifier (i.e.,"class foo<int>") (C++) 259

E2506: Explicit specialization of 'specifier' is ambiguous: must specify template arguments (C++) 257

E2507: 'class' is not a direct base class of 'class' (C++) 268

E2508: 'using' cannot refer to a template specialization (C++) 279

E2509: Value out of range (C++) 258

E2510: Operand size mismatch (C++) 258

E2511: Unterminated macro argument (C++) 263

E2512: Cannot emit RTTI for return type of 'function' (C++) 267

E2513: Cannot emit RTTI for 'parameter' in 'function' (C++) 267

E2514: Cannot (yet) use member overload resolution during template instantiation (C++) 279

E2515: Cannot explicitly specialize a member of a generic template class (C++) 265

E2522: Non-const function 'function' called for const object (C++) 267

E2523: Non-volatile function 'name' called for volatile object (C++) 267

E2524: Anonymous structs/unions not allowed to have anonymous members in C++ (C++) 264

E2525: You must define _PCH_STATIC_CONST before including xstring to use this feature (C++) 257

E2526: Property 'name' uses another property as getter/setter; Not allowed (C++) 257

E2527: Option 'name' cannot be set via 'name' (C++) 258

E2528: Option 'name' must be set before compilation begins (C++) 258

E2529: Path 'path' exceeds maximum size of 'n' (C++) 268

E2530: Unrecognized option, or no help available (C++) 258

E2531: Parameter is an incomplete type (C++) 383

E2532: Constant in new expression requires an initializer (C++) 382

E2533: Parameter mismatch (wanted typename) (C++) 382

E2534: Integral constant expression expected (C++) 382

E2535: Incomplete type cannot be part of a throw specification (C++) 382

E2536: Incomplete type cannot be part of a exception declaration (C++) 381

E2537: Cannot create instance of abstract class (C++) 380

E2538: Static assert failed: '%s' (C++) 381

E2539: Constant expression expected (C++) 383

E2540: String literal expected (C++) 383

E2541: Attribute '%s' cannot be set (C++) 382

E2542: '%s' is marked 'final' and cannot be overriden (C++) 383

E2543: Combination of options 'options' is not permitted (C++) 381

E2544: Function exception specifications do not match (C++) 381

E2545: Enum underlying type must be an integral (C++) 382

E2546: Redeclaration of enum is not identical (C++) 382

E2547: ... expected (C++) 383

E2548: ... was unexpected; expression is not a variadic template pattern (C++) 381

E2549: Operand is not a parameter pack (C++) 381

E2550: No arguments can follow a variadic template in an argument list (C++) 381

E2551: Return statement not allowed in __finally block (C++) 384
E2552: This feature is not (yet) supported (C++) 383
E2553: %s mismatch in redeclaration of '%s' (C++) 383
ecvt 1112
edit controls
displaying data 1770
text controls 2049
Edit Controls 2049
Editing an Existing Package 2216
Editing and Updating Data 1773
Editing Code 1936
editing data
in grids 1760
Editing Data 1715
Editing Data in a Control 1773
Editing Delta Packets Before Updating the Database 1813
Editing in the Grid 1760
Editing Menu Items in the Object Inspector 1983
Editing Package Source Files Manually 2216
editing properties
properties 1349
Editing Records 1591
Editing the Main Form 2256
Editing the Property as a Whole 1349
EDOM, ERANGE, #defines 832
Elements of an ActiveX Control 1408
Eliminating Pointers In Templates 412
Embedding Tables in HTML Documents 2264
Empty Handlers Must Be Valid 1240
Emptying Tables 1592
Enabling and Disabling Action Items 2279
Enabling and Disabling Filtering 1593
Enabling and Disabling the Data Source 1774
Enabling Application Code 1924
Enabling BDE-based Cached Updates 1688
Enabling C++ Applications for Unicode 507
Enabling Casting 2097
Enabling Editing in Controls On User Entry 1774
Enabling Help in applications 1887
Enabling Help in Applications 1893
Enabling Mouse, Keyboard, and Timer Events 1774
Enabling Simple Data Binding with the Type Library 1414
Encapsulating Graphics 1319
Ending a Drag Operation 2162
enum 548
keywords, C++ 548
enum, assignment
language structure, C++ 618
Enumeration Constants 673
enumerations
language structure, C++ 617
Enumerations 617
eof 898
EOF #define 1030
Equality Operators 590
ERC.EXE 213
errno (C Runtime Library Reference) 833
errno.h 831
Error Events 2342
Error Numbers in errno 834
Errors And Overflows 624
escape sequences
lexical elements, C++ 664
Escape Sequences 664
Establishing Master/detail Relationships Using Parameters 1593
Evaluation Order 624
Event handler 1239
Event Handler Types Are Procedures 1236
event handlers
declaring events 1240
deleting 2145
events 1238
Event Handlers Are Optional 1237
Event Handlers Have A Return Type of void (C++) 1236
Event Types Are closure Types (C++) 1236
Event Types Are Method-pointer Types 1235
events 1234
Automation controllers 1425

creating 1233
default 2147
handling 2145, 2146, 2147, 2148, 2149
menu 2144
OnUpdateRecord 1693
triggering events 1239
types 2057
user-defined events 1238
Events Are closures (C++) 1234
Events Are Method Pointers 1234
Events Are Properties 1235
Examining a Delphi Object 2067
Example for "Temporary used ..." error messages (C++) 361
Example of Adding Audio and/or Video Clips (VCL Only) 2179
Example of Adding Silent Video Clips 2180
Example of Declaring Methods 1243
Example Of Overloading Operators 439
Example: Message Handler 1306
Example: Message Structure 1305
Example: Overriding a Message Handler 1307
Example: User-defined Messages 1304
except.h 838
Exception Errors 5
exception handlers
keyword 2018
reraising 2020
scope 2021
exception handling
compiler options 2016
constructors 2017
exceptions 2017, 2018
safe pointers 2021
VCL 2029, 2030
Exception handling 2014
Exception Handling 2018
Exception Handling Options (C++) 2016
exception objects
classes 2027
defining 2029
VCL 2028
Exception-handling Statements 2018
exceptions
finally blocks 2025
handlers 2024
raising 2020
silent 2022
throwing 2019
try blocks 2026
exceptions, unwinding 2016
Excluding Blank Cells 1265
execl, execle, execlp, execlpe, execv, execve, execvp, execvpe, _wexecl, _wexecle, _wexeclp, _wexeclpe, _wexecv, _wexecve, _wexecvp, _wexecvpe 987
executable files
internationalizing 1933, 1934
Executing a Batch Move 1680
Executing a Search with Find Methods 1594
Executing a Search with Goto Methods 1595
Executing an Update Statement 1639
Executing Commands That Do Not Return Records 1829
Executing Queries That Don't Return a Result Set 1596
Executing SQL in the Data Explorer 74
Executing Stored Procedures That Don't Return a Result Set 1596
Executing the Command 1829
Executing the Commands using TSQLDataSet 56
Executing the SQL Statements 1643
Executing thread objects 2238
Executing Thread Objects 2242
exit 1114
Exit And Destructors 423
EXIT_xxxx #defines 1095
exp, expl 944
explicit 548
keywords, C++ 548
Explicit
implicit rules (MAKE) 193
Explicit Access Qualification 428
Explicit Conversion Operators (C++0x) 493

export 549
export,
keywords, C++ 549
Exporting And Importing Templates 432
Expression Statements 647
expressions
language structure, C++ 620, 623
Expressions 619
Expressions (C++) 620
Expressions And C++ 623
expressions, errors
language structure, C++ 624
expressions, evaluation order
language structure, C++ 624
Extended Types Formatted I/O 634
extending
namespaces, C++ 428
Extending A namespace 428
Extending the IDE 1276, 1285
Extending the Interface of the Application Server 1531
extern 549
keywords, C++ 549
extern Templates (C++0x) 494
External Declarations and Definitions 606

F

F1000: Compiler table limit exceeded (C++) 285
F1001: Internal code generator error (C++) 315
F1002: Unable to create output file 'filename' (C++) 284
F1003: Error directive: 'message' (C++) 284
F1004: Internal compiler error (C++) 284
F1005: Include files nested too deep (C++) 286
F1006: Bad call of intrinsic function (C++) 284
F1007: Irreducible expression tree (C++) 285
F1008: Out of memory (C++) 286
F1009: Unable to open input file 'filename' (C++) 285
F1010: Unable to open 'filename' (C++) 286
F1011: Register allocation failure (C++) 285
F1012: Compiler stack overflow (C++) 285
F1013: Error writing output file (C++) 285
FA_xxxx #defines 822
fabs, fabsl 945
fastmath.h 842
fclose 1030
fcntl.h 844
fcvt 1114
feof 1033
ferror 1034
Fetching Metadata into a dbExpress Dataset 1830
Fetching Multiple Result Sets 1597
Fetching Records Asynchronously 1481
Fetching the Data 1830
Fetching the Data using TSQLDataSet 57
fflush 1035
fgetc, fgetwc 1036
fgetpos 1037
fgets, fgetws 1039
field attributes
removing 1860
field datalink class 1338
field objects
dynamic vs. persistent 1876
events 1871
fields 1877
methods 1864
properties 1860, 1871, 1873
fields
default values 1860
restricting input 1878
updating values 1773
File Date-time Routines 2125
File dumping utility 204
File Inclusion With #include 692
File search utility
GREP.EXE 170
file stream
VCL Forms 136
filelength 899

files 2142
copying 2123
date-time routines 2125
deleting 2124
finding 2125
manipulating 2127
reading and writing 2091
renaming 2130
TFileStream 2133
Filling in the Cells 1265
Filling in the Response Header 2284
Filtering Multiple Rows Based On Update Status 1481
Filtering Records Based On Bookmarks 1482
filters
bookmark-based 1482
client datasets 1741
ranges 1615
specifying 1608, 1635
final 550
final attribute
C++0x features 492
Find method
copy stream 138
findclose, _wfindclose 804
findfirst, _wfindfirst 805
Finding a File 2125
findnext, _wfindnext 807
Fine-tuning a Connection 1483
flat files
loading 1739
saving 1740
FlipChildren Method 1927
float 552
keywords, C++ 552
Float and Double Limits 1216
float.h 853
Floating Functions 502
Floating functions (C++) 502
Floating Point Constants 672
Floating point error: Divide by 0 OR Floating point error: Domain OR Floating point error: Overflow (C++) 312
Floating point error: Partial loss of precision OR Floating point error: Underflow (C++) 312
Floating point error: Stack fault (C++) 312
floor, floorl 946
fmod, fmodl 946
fnmerge, _wfnmerge 808
fnsplit, _wfnsplit 810
fonts
deploying 1950
Fonts 1950
fopen, _wfopen 1041
for 552
keywords, C++ 552
Forcing Asynchronous Connections 1483
Formal Parameter Declarations 635
formats
internationalizing 1930
Formats and Sort Order 1930
formatting data
data formats 1873
Forming the Connection 2347
forms
adding 1965
creating 1990
creating dynamically 1980
creating modeless 1981
displaying an auto-created 1983
layout 1988
memory management 1975
passing additional arguments 1993
retrieving data from 1997, 1999
retrieving data from modal 1997
using a local variable to create an instance 2010
Forward Declaration of Enums (C++0x) 495
fprintf, fwprintf 1042
fputc, fputwc 1043
fputs, fputws 1044
frame

published properties 2147
frames 2012
creating 1977
sharing 2005, 2008
fread 1045
free 722
freopen, _wfreopen 1046
frexp, frexpl 947
friend 553
keywords, C++ 553
Friends Of Classes 388, 389
friends, of classes
C++ language specifics 389
fscanf, fwscanf 1047
fseek 1048
fsetpos 1049
fstat, stat, _wstat 1189
ftell 1050
ftime 1191
Function Call Operator 644
function calls 696
language structure, C++ 635
Function Calls And Argument Conversions 635
Function Failure Errors 6
Function Modifiers 616
Function Templates 433
Function Templates Overview 433
functions
conversion 2110, 2113
language structure, C++ 634
Functions 624
functions, declarations
language structure, C++ 625, 633
functions, parameters
language structure, C++ 635
Fundamental WebSnap components 2321
Fundamental WebSnap Components 2326
fwrite 1051

G

gcvt 1115
general applications
deploying 1949
Generating a Handler for a Component's Default Event 2147
Generating a New Event Handler 2146
Generating an ActiveX Control Based On a VCL Form 1410
Generating the content of response messages 2261
Generating the Content of Response Messages 2269
Generating the Day Numbers 1266
Generating WSDL Documents for a Web Service Application 2306
Generics
C++ handling 511
geninterrupt 826
GenTLB.exe 1468
getc, getwc 1052
getch 747
getchar, getwchar 1052
getche 748
getcurdir 11
getcurdir, _wgetcurdir 811
getcwd, _wgetcwd 812
getdate, setdate 826
getdfree 827
getdisk, setdisk 813
getenv, _wgetenv 1116
getftime, setftime 899
getpass 749
getpid 989
gets, _getws 1053
gettext 750
gettextinfo 751
gettime, settime 828
Getting Information About the Client or Server Socket Connection 2351
Getting Information About the Client Socket Connection 2347
Getting more information 1555

Getting More Information 1555
Getting Parameters from the Application Server 1715
Getting Started with InterBase Express 17
Getting started with IntraWeb 2254
Getting Started with IntraWeb 2255
Getting Started with Rave Reports 1555
Getting the Pen Position 2192
global routines
    helper objects 2090
gmtime 1203
goto 553
    keywords, C++ 553
gotoxy 752
Granting Permission to Access and Launch the Application Server 1532
Graphic Controls 2040
Graphic Images 1930
graphics 2176
    adding to controls 2154
    canvases 1319
    copying to clipboard 2186
    cutting to clipboard 2186
    displaying 2040
    drawing on 2187
    internationalizing 1930
    object types 2204
    overview 2208
    pasting from clipboard 2195
    scrollable 2195
    shapes 2047
    using the clipboard 2206
    VCL Forms applications 93
graphics files
    loading and saving 2194
GREP.EXE 170
GREP.EXE, the text search utility 170
grids
    draw grids 2040
    non-database 2049
    string grids 2049
    value list editors 2036
Grids 2049
group boxes
    radio groups 2041
Group Boxes and Radio Groups 2041
Grouping Controls 2041
GUI applications 1899
GUI Applications 1902
Guidelines for Using Decision Support Components 1804

H

Handling Batch Move Errors 1684
Handling Boolean Field Values with Check Boxes 1775
Handling Cached Update Errors 1696
Handling Classes of Exceptions 2027
Handling Command Parameters 1484
Handling Constraints from the Server 1716
Handling Delphi Features in C++Builder 2009 507
Handling Errors For The New Operator 435
Handling Events 1871
Handling Events in an Automation Controller 1425
Handling Exceptions 129
Handling Exceptions in the Thread Function 2234
Handling Exceptions in VCL Applications 2030
Handling messages 1298
Handling Messages and System Notifications: Overview 1308
Handling Mouse-down and Key-down Messages 1332
Handling Multiple Drawing Objects in Your Application 2192
Handling Palettes 1378
Handling Server Constraints 1814
Handling the OnPopup Event 2163
Handling the Screen 1985
header controls 2042
Header Controls 2042
Header File Search With "header_name" 693
Header File Search With <header_name> 693
header files 693
headers
    HTTP 2252

SOAP 2303, 2310
heapcheck 723
heapcheckfree 724
heapchecknode 725
heapfillfree 726
heapwalk 727
hello world
VCL Forms 101, 106, 136
Help and Hint Properties 2042
Help files for components 1347
Help system 1898
customizing 1891
Help Manager 1897
Help viewers 1890, 1892, 1893, 1894
IHelpSystem 1898
Man pages 1893, 1895
TApplication 1890
TApplication (VCL) 1895
TControl (VCL) 1895
Help System Interfaces 1894
helper applications 1950
Helper Applications 1950
HIBITx #defines 1215
Hiding 432
Hiding and Showing Toolbars 1987
Hiding Implementation Details 1374
hiding names
C++ language specifics 432
Hiding the Main Form 1986
Hiding Unused Items and Categories in Action Bands 1986
highvideo 752
hints
help 2042
host environments
programming 1953
hot key controls 2043
Hot Key Controls (VCL Only) 2043
How Actions Find Their Targets 1987
How Delphi Adds Events 1413
How Delphi Adds Properties 1412
How TApplication Processes VCL Help 1895
How To Construct A Class Of Complex Vectors 438
How to Handle Delphi Anonymous Methods in C++ 508
How to Handle Delphi AnsiString Code Page Specification in C++ 510
How to Handle Delphi Generics in C++ 511
How VCL Controls Process Help 1895
HTML
producing 2270
HTML commands
database information 2265
HTML documents
databases and 2266
HTML tag editor
editing HTML tags 151
HTML Templates 2269
HTML-transparent tags 2271
HTTP
advantages 1550
overview 2248
requests 2249
HTTP messages
content 2276, 2285
headers 2274, 2284
processing 2281
responding to 2283
response content 2269
sending 2285
types 2280
HTTP Request Header Information 2252
HTTP request messages
TWebRequest object 2274
HTTP server activity 2247
HTTP Server Activity 2248
HUGE_VAL #defines 931
hypot, hypotl 948

I

IAppServer interface
remote data modules 1811
IDE
adding actions 1278
adding images 1279
deleting toolbar buttons 1284
extending 1285
responding to IDE events 1289
identifiers
lexical elements, C++ 679
Identifiers 679
Identifiers Overview 679
Identifying Application Files 1950
Identifying Standard Events 1237
Identifying the Database 1648
IDL
type library syntax 1455
if
keywords, C++ 553
if, else 553
IInterface interface
interfaces:deriving 2075
ILINK32 191, 197
ILINK32.EXE 174
ILINK32.EXE, the Incremental Linker 174
image control adding 2181
images
adding to a string list 2154
adding to an application 2155
data-aware 1771
displaying 2043
Images 2043
IMPDEF 179
IMPDEF.EXE 179
IMPDEF.EXE, the Module Definition Manager 179
Implementing Binary Operations 2100
Implementing COM Objects with Wizards 1397
Implementing Commands 1349
Implementing Comparison Operations 2102
Implementing Drag and Dock in Controls 2163
Implementing Drag and Drop in Controls 2160
Implementing ICustomHelpViewer 1895
Implementing IExtendedHelpViewer 1896
Implementing IHelpSelector 1897
Implementing IInterface 2075
Implementing services 2338
Implementing Services 2338
Implementing the Standard Events 1237
Implementing the Wizard Interfaces 1286
Implementing Unary Operations 2103
implements keyword
interfaces 2078
IMPLIB.EXE 181
IMPLIB.EXE, the Import Library Tool 181
Implicit And Explicit Template Functions 434
Import library tool (C++) 181
import, _import, __import 554
Importing Data from Another Table 1670
Importing Resource Files 1988
Importing Type Library Information 1420
Importing WSDL Documents 2307
Include files (C++) 182
Including Bi-directional Functionality in Applications 1926
Including the Form Unit 1361
Incomplete Declarations 653
Increment/decrement Operators 643
indexes
listing 1603
searching for records 1615
sorting records 1609
specifying alternative 1630
Indicating That a Control Is Owner-drawn 2166
Indicating the Need for Client Action 2284
Indicating the Response Status 2285
Indicating the Types of Operations the Connection Supports 1485
Indicating What Records Are Modified 1717

Influencing How Updates Are Applied 1815
Informational messages (C++) 284
inheritance
objects 2069
Inheriting an Item 1915
Inheriting Data and Code from an Object 2069
initialization
language structure, C++ 611
Initialization 611
Initializing a Thread 130
Initializing After Loading 1256
Initializing enumeration with type (C++) 360
Initializing Owned Classes 1225
Initializing the Data Link 1332
Initializing the Thread 2234
inline 554
keywords, C++ 554
inline functions
C++ language specifics 391
keywords, C++ 525
Inline Functions 391
In-process, Out-of-process, and Remote Servers 1389
input controls
controls 2047
input method editor
IME 1928
insline 753
Inspecting the Update Status of Individual Rows 1485
installation programs
InstallAware Express 1954
Installing Component Packages 2217
Installing the Wizard Package 1288
installing unit into a package 97
int 555
keywords, C++ 555
INT_xxx #defines 913
Integer Constant Without L Or U 669
Integer Constants 674
InterBase components
getting started 17
Interface Version Numbers 1288
interfaces 2079
Automation 1438
dispatch 1438
invokable 2292
polymorphism 2077
properties 1253
remote data modules 1531
reusing code 2076
TInterfacedObject 2072
Internal Data Storage 1251
Internal Representation of Numerical Types 675
Internal Representation Of Numerical Types 676
international applications 1923
bi-directional 1926, 1927, 1928
localizing 1934
internationalization
definition 1923
Internationalization 1923, 1924
Internationalization and Localization 1923
Internationalizing Applications 1924
Internet palette page
Web server applications 2251
InternetExpress page producers
templates 1529
Web pages 1529
Interoperable Applications Procedures 79
Intervening as Updates Are Applied 1718
IntraWeb
VCL for the Web 2255, 2256, 2257, 2258, 2259, 2260
Introduction to component creation 1310
Introduction To Constructors And Destructors 415
invokable interfaces
calling 2307
non-scalar types 2293, 2294
Invoking Destructors 424
io.h 864
IP addresses

hosts vs. 2336
isalnum, __iscsym, iswalnum, _ismbcalnum 774
isalpha, __iscsymf, iswalpha, _ismbcalpha 775
ISAPI applications
debugging 2244
isascii, iswascii 776
isatty 901
iscntrl, iswcntrl 777
isdigit, iswdigit, _ismbcdigit 778
isgraph, iswgraph, _ismbcgraph 779
islower, iswlower, _ismbclower 779
Isolating Resources 1931
isprint, iswprint, _ismbcprint 780
ispunct, iswpunct, _ismbcpunct 781
isspace, iswspace, _ismbcspace 782
isupper, iswupper, _ismbcupper 783
isxdigit, iswxdigit 784
Iterating Through a Session's Database Components 1652
Iterating Through Strings in a List 123, 2138
Iteration Statements 648
itoa, _itow 1117

J

javascript libraries 1549
Jump Statements 648

K

kbhit 754
Keeping Track of Which Drawing Tool to Use 2193
Keyboard Mappings 1931
key-down messages 1335
Keyword Extensions 681
keywords 690
language structure, C++ 607
lexical elements, C++ 680, 681
Keywords 680
Keywords And Protected Words In Macros 690
Keywords Overview 680
Keywords, Alphabetical Listing 513
Keywords, By Category 579

L

L_ctermid #define 1024
L_tmpnam #define 1024
Labeled Statements 648
labels
displaying data 1772
static text controls 2043
Labels 2043
labs 1118
Language Structure 586
late_check 576
keywords, C++ 576
ldexp, ldexpl 949
ldiv 950
lexical elements
lexical elements, C++ 661
Lexical Elements 660, 661
lfind 1119
Library manager (TLIB.EXE) 208
license requirements
software license requirements 1954
Licensing ActiveX Controls 1410
Limiting Dimension Selection in Decision Grids 1790
Limiting Records with Parameters 1720
Limiting Records with Ranges 1597
Limiting What Records Appear 1741
limits.h 913
line continuation 695
line drawing 2187
refining 2196
lines and polylines drawing 2187
linkage
language structure, C++ 595
Linkage 595
Linker (C++) 174
Linking DLLs 1883
list boxes 2044

data-aware 1778
data-aware data 1768
List Boxes and Check-list Boxes 2044
List Controls 2042
list views 2044
List Views 2044
listening connections 2345
Listening Connections 2345
Listening for Client Requests 2350
lists 2139
controls 2042
operations 2091
persistant 2105
string 2140
Loading a Picture from a File 2194
Loading and Saving Custom Variant Values 2104
Loading and Saving Graphics Files 2194
Loading and Saving String Lists 2138
Loading and Storing Graphics 1378
Loading Data from a File or Stream 1739
Loading Data from and Saving Data to Files 1486
loading images
files 1378
Loading Packages in an Application 2221
Loading Packages with the LoadPackage Function 2211
locale.h 915
localeconv 915
Locale-specific Features 1928
localization
definition 1924
Localization 1924
Localization Considerations in UI Design 1929
Localizing Applications 1934
localtime 1204
Locating Event Handlers 2147
Locating Items in a String List 2138
lock 902
locking 903
locking objects 2227
Locking Objects 2227
log, logl 950
log10, log10l 951
logical operators
operators, C++ 590
Logical Operators 590
Login Pages 2329
Login support 2327
Login Support 2330
logins
requiring 2330
long 555
long string routines
strings 2118
LONG_xxx #defines 914
longjmp 994
lookup fields
defining 1868
lookup list boxes
lookup combo boxes 1769
lowvideo 755
lsearch 1120
lseek 904
ltoa, _ltoa, _ltow 1121

M

M_E, M_LOGxxx, M_LNxx #defines 932
M_SQRTxx #defines 932
macros
predefined (C++) 708
Macros With Parameters 694
Macros With Parameters Overview 694
main form
hiding 1986
MAKE 183
MAKE Directives 186
MAKE macros 191
MAKE Macros 191
MAKE Rules (Explicit and Implicit) and Commands 193

MAKE.EXE 183
Making a Control a Dockable Child 2164
Making a control data aware 1325
Making a Control Data Aware 1333
Making a dialog box a component 1358
Making a Dialog Box a Component: Overview 1362
Making a Windowed Control a Docking Site 2164
Making Component Help Context-sensitive 1350
Making components available at design time 1339
Making Components Available at Design Time: Overview 1350
Making Events Visible 1238
Making Methods Virtual 1245
Making Scrollable Graphics 2195
Making Source Files Available 1323
Making the Control Read-only 1333
Making the Table a Detail of Another Dataset 1598
malloc 728
malloc.h 920
Managing Database Connections 1665
Managing database sessions 1649
Managing Database Sessions 1664
Managing Database Sessions Using TSession 67
Managing Events in Your Automation Object 1437
Managing Layout 1988
Managing Multiple Sessions 1659
Managing Server Connections 1533
Managing Transactions 1498
Managing Transactions in Multi-tiered Applications 1533
Manipulating Files 2127
Manipulating Menu Items at Runtime 1989
Manipulating Strings in a List 2139
Mapping Between XML Nodes and Data Packet Fields 1843
Mapping Data Types 1684
Marking and Returning to Records 1599
marshaling
    mechanism 1390
Marshaling Data 1439
master
    detail relationships/multi-tiered applications 1539
    detail relationships/queries 1593
    detail relationships/tables 1590
    detail tables/unidirectional datasets 1833
math.h 921
max 1122
MAXxxxx #defines (fnsplit) 804
MAXxxxx #defines (integer data types) 1215
mblen 1122
mbstowcs 1123
mbtowc 1124
MDI applications
    multiple document interface 1901
    SDI applications 1902
MDI Applications 1901
measurements
    adding units 2091, 2095
mem.h 959
Member Access Control 399
member functions
    C++ language specifics 405
Member Functions 405
Member Scope 396, 400
members, scope
    C++ language specifics 400
memccpy 962
memchr, _wmemchr 963
memcmp 964
memcpy, _wmemcpy 965
memicmp 965
memmove 966
Memo and Rich Edit Controls 2044
memo fields
    displaying and editing 1771
    displaying with format information 1772
Memory Block Comparison Warnings 11
memory management
    interfaced objects 2075, 2076, 2081
Memory Management of Interface Objects 2075
memset, _wmemset 967

Menu Designer
context menu 2010
opening 1992
menu items
disabling 2160
menus
accelerator keys 2005
adding images 1967
adding menu items 1968
building 1973
color and bitmaps 1966
creating 1975
creating submenus 1978
customizable 1976
editing menu items 1983
icons 1966, 2001
manipulating menu items at runtime 1989
merging 1982, 1989
moving menu items 1990
naming 1991
naming menu items 1991
specifying the active menu 2006
switching between 2006
templates 1990, 2001, 2009
VCL Forms 100, 102, 110, 113
viewing 2011
merge modules
packages 1951
Merge Modules 1951
Merging Changes into Data 1720
Merging Menus 1989
message handlers
creating 1304
overriding message handlers 1307
parameters 1309
Windows messages 1304
Message Options 197
messages
dispatching messages 1306
overview 1308
trapping messages 1308
Messages (C+) 197
metadata
dbExpress 1824, 1830
methods 2145
dispatching 1373
dynamic 1373
functions 1243
nonvirtual 1375
static 1375
virtual methods 1374
Methods That Should Be Protected 1244
Methods That Should Be Public 1244
min 1125
Mixed-Language Calling Conventions 615
Mixing and Converting String Types 2127
mkdir, _wmkdir 814
mktime 1206
modal
VCL Forms 120
modal form
event handler 120
modeless
event handler 121
VCL Forms 121
modf, modfl 952
modifiers
keywords, C++ 571
Modifiers 582
modifiers, function
language structure, C++ 616
Modifying a Range 1601
Modifying an existing component 1363
Modifying an Existing Component: Overview 1365
Modifying an Interface Using the Type Library 1461
Modifying Connections in the Data Explorer 75
Modifying Data 1629
Modifying Entire Records 1601

Modifying Existing Controls 1316
Modifying Shared Items 1916
Modifying the Change Method 1334
Modifying the Component Object 1365
Module definition files 199
Module Definition Files 199
Module Definition Manager
    IMPDEF.EXE 179
mouse
    responding to 2199
mouse actions
    adding a field to a form 2181
mouse events
    data controls 1774
    parameters 2207
mouse move
    responding to 2198
mouse pointer
    drag-and-drop 2157
mouse-down action 2198
mouse-down messages 1336
    key-down-messages 1332
mouse-up action 2199
MoveFile
    RenameFile 137
movement
    tracking 2202
movetext 755
Moving Menu Items 1990
Moving the Selection 1268
MTS
    database applications 1546
Multibyte Character Sets (MBCS) 1925
Multidimensional Crosstabs 1797
multimedia 2177
Multiplicative Operators 593
multi-read exclusive write synchronizer
    synchronizer 2228
Multithread Programs 629
Multithread Variables 616
multithreaded applications
    checking other threads 132
    cleaning up threads 125
    critical sections 125
    default priorities 130
    exception handling 129
    Execute method 134
    initializing threads 130
    locking objects 125
    main thread 131
    multi-read exclusive-write synchronizer 125
    simultaneous thread access 125
    synchronizing background threads 131
    task completion 132
    thread object 126
    thread-local variables 131
    waiting for threads 132
    writing the thread function 134
multi-threaded applications
    applications 1659
multi-tiered applications
    ActiveX clients 1530
    advantages 1519
    client applications 1518
    connection protocols 1523
    creating 1520
    data providers 1819
    InternetExpress 1522
    multiple remote data modules 1526, 1547
    overview 1534
    providers 1544
    state information 1540
multi-tiered database applications
    client datasets 1564
mutable 555
    keywords, C++ 555

N

namespace 556
namespace Alias 429
namespaces, overview
    C++ language specifics 556
naming
    threads 2239, 2241
Naming a Data Module and Its Unit File 1912
Naming a Session 1659
Naming a Thread 2241
naming conventions
    methods 1244
Naming Conventions for Template Menu Items and Event Handlers 1990
Naming Menus 1991
Naming Methods 1244
Naming the Menu Items 1991
Navigating and Manipulating Records 1767
Navigating Data in Client Datasets 1721
Navigating Datasets 1634
Navigating Days 1269
Navigating Months and Years 1269
Navigating Records in a Filtered Dataset 1603
NDEBUG #define 731
nested types
    C++ language specifics 401
Nested Types 401
nesting 695
Nesting Parentheses And Commas 695
new 556
    alternative representations of C++ tokens 575
    C++ language specifics 556
New Field dialog box 1855
new.h 968
New-style Typecasting 438
New-style Typecasting Overview 438
NFDS #define 822
Non-blocking connections 2340
Non-blocking Connections 2340
nonvisual components
    creating 1318
noreturn 557
noreturn attribute
    C++0x feature 492
normvideo 756
not
    alternative representations of C++ tokens 557
Not Using Reference Counting 2076
not, ! 557
not_eq
    alternative representations of C++ token 558
not_eq, != 558
Notifying a Wizard of IDE Events 1289
NULL #define 1005
null directive 711
Null pointer assignment (C++) 360
nullptr 558
    keywords, C++ 558
null-terminated string routines
    strings 2121

O

O_xxxx #defines 845
O2237: DPMI programs must use the large memory model (C++) 318
object fields
    fields 1862
Object List Pane 1448
object pooling
    remote data modules 1535
Object Repository
    Repository 1915, 1916, 1917
object-oriented programming 2071
    component writing 1369
Object-oriented programming for component writers 1367
Object-oriented Programming for Component Writers: Overview 1369
objects

classes 2071
components 2057
creating 2066
Delphi 2067
language structure, C++ 596
variables 2064
Objects 596
Objects, Components, and Controls 2057
Obtaining Aggregate Values 1721
Obtaining an Editable Result Set 1676
Obtaining Information About Indexes 1603
Obtaining Metadata 1501
Obtaining Tools API Services 1293
OEM and ANSI Character Sets 1925
OEM character sets
ANSI character sets 1925
Off-screen Bitmaps 1379
offsetof 1006
OLE containers
object linking and embedding 1426
One-Dimensional Crosstabs 1795
OnPopup event
handling 2163
Open Tools API
Tools API 1294
open, _wopen 849
OPEN_MAX #define 1024
opendir, wopendir 795
Opening a Connection Using TDataBase 1646
Opening an Existing Type Library 1460
Opening and Closing Datasets 1604
Opening and Closing Decision Grid Fields 1790
Opening Database Connections 1653
Opening the Dataset in Batch Update Mode 1487
Opening the Menu Designer 1992
Operating System Versions 1955
operator 558
keywords, C++ 558
operators, C++ 592, 643
operator new
C++ language specifics 436, 437
Operator new 437
Operator new Placement Syntax 436
Operator Overloading Overview 438
Operators 583
Operators Summary 635, 636
operators, binary
operators, C++ 587
operators, C++ specific
operators, C++ 590
operators, equality
language structure, C++ 590
operators, expressions
operators, C++ 645
operators, function calls
operators, C++ 644
operators, overloading
operator overloading, C++ 439, 441, 442, 443, 444, 445
operators, relational
in C++ 591
operators, summary
operators, C++ 636
operators, unary
operators, C++ 656
or
alternative representations of C++ tokens 559
or, || 559
or_eq
alternative representations of C++ token 559
or_eq, |= 559
Order Of Calling Constructors 420
Organizing Actions for Toolbars and Menus 1992
Other Techniques for Sharing Memory 2226
Overloaded functions, resolution 458
Overloaded Operators And Inheritance 444
Overloading Binary Operators 442
Overloading Constructors 421
Overloading Operator Functions 444

Overloading Operator Functions Overview 441
overloading operators
C++ language specifics 436
Overloading Operators 441
Overloading The Assignment operator = 442
Overloading The Class Member Access Operators -> 443
Overloading The Function Call Operator ( ) 443
Overloading The Operator delete 436
Overloading The Operator new 436
Overloading The Subscript Operator [] 445
Overloading Unary Operators 445
Overriding A Template Function 434
overriding methods
methods 1374
Overriding Methods 1374
Overriding the Constructor 1365
Overriding the Constructor and Destructor 1226
Overriding the Dataset On the Application Server 1722
Overriding the Default Priority 2242
Overriding the DefineProperties Method 1257
Overriding the Handler Method 1307
Overview of a Three-tiered Application 1534
Overview of ADO Components 1487
Overview of COM Technologies 1385
Overview of Component Creation 1313
Overview of Decision Support Components 1790
Overview of Graphics 1379
Overview of Graphics Programming 2176
Overview of the Tools API 1294
Overview of Using Cached Updates 1723
owner-draw controls 2161, 2166, 2167

P

P_xxxx #defines 975
package
installing units into 97
Package Collection Files 2217
Package Files 1952
Package Files Created by Compiling 2218
packages
and DLLs 2219
compiler directives 2219
compiling 2212
creating 2213
custom 2214
deploying 1952, 2215, 2217
design-time 2215
editing an existing 2216
editing source files manually 2216
files 2218
installing component 2217
loading/unloading 2211
overview of creating and editing 2213
packages 2211
runtime defined 2220
structure 2220
using command-line compiler and linker 2222
using in an application 2221
weak packaging 2223
when to use 1883
which runtime to use 2214
why use 2224
packages and DLLs
creating 1882
Packages and Standard DLLs 2219
Package-specific Compiler Directives 2219
page controls 2045
Page Controls 2045
Page dispatcher operation 2321
page options
WebApp 148
page producers
chaining 2268
Page Producers 2326
Pages of Type Information 1449
paint boxes
drawing 2045
Paint Boxes 2045

palettes
graphics 1378
panels 2046
Panels 2046
Paradox tables
local transactions 1698
network control files 1654
passwords 1662
parameters
ADO command 1484
getting from providers 1715
ParentBiDiMode Property 1927
Parts of a COM Application 1386
Parts of a Uniform Resource Locator 2253
Parts of the Type Library Editor 1446
pascal
keywords, C++ 559
pascal, _pascal, __pascal 559
Passing a Local Variable as a PChar 2128
Passing Additional Arguments to Forms 1993
Passing File Information To Child Processes 630
Passing Parameters to the Source Dataset 1723
Pasting Graphics from the Clipboard 2195
Pathname Merging and Splitting Warnings 11
paValueList
paSubProperties 1354
PChar local variables 2128, 2131
pen color 2182
pen mode 2182
pen position 2192
pen style 2182
pen width 2183
pens 2206
Perform method
sending messages 1302
Performing Conversions 2105
perror,_wperror 835
persistent columns
adding buttons 1761
creating 1756
deleting in Columns editor 1757
reordering 1751
persistent connections 1661
Persistent Field Components 1859
persistent fields
creating 1874
defining 1856
deleting 1857
field objects 1859
ordering 1854
Persistent Lists 2105
PI constants 932
picture
loading from a file 2194
saving to file 2200
pictures
graphics 1381
replacing 2197
pixels
reading and setting 2196
Placing A Bitmap Image in a Control in a VCL Forms Application 135
Placing and Naming Components 1913
placing bitmap images 135
Placing the Control 2196
Plus And Minus Operators 657
Pointer Arithmetic 639
Pointer Constants 640
Pointer Conversions 640
Pointer Declarations 641
pointers
classes 1374
language structure, C++ 641, 642
Pointers 637, 641
Pointers To Functions 642
Pointers To Objects 642
pointers, constants
language structure, C++ 640
pointers, conversions

language structure, C++ 640
pointers, objects
language structure, C++ 639
poly, polyl 953
polygons 2189
polylines 2189
polymorphic classes
C++ language specifics 447
Polymorphic Classes 446, 447
Pooling Remote Data Modules 1535
popup menus 2164
Portable code (C++) 502
ports 2337
services and 2339
Possible Declarations 603
Postfix Expression Operators 642
Posting Data 1606
PostMessage method
sending messages 1303
pow, powl 954
pow10, pow10l 955
Pragma Directives Overview 696
precedence
operators, C++ 622
Precedence Of Operators 622
Precompiled header files 203
Predefined Action Classes 1995
Predefined Macros 708
Predefined Macros Overview 707
Preparing Queries 1606
Preparing Stored Procedures 1607
Preprocessor (C
C++) 167
Preprocessor Directives 710
Primary Expression Operators 644, 645
printf, wprintf 1055
Printf/Scanf floating-point formats not linked (C++) 361
Printing 2129
private 560
keywords, C++ 560
Private, Protected, Public, and Published Declarations 2069
procedure
Register 1357
procedures
interfaces 2080
Procedures 47
process.h 971
Processing Headers in Client Applications 2310
Programming a Calculated Field 1870
Programming for Varying Host Environments 1953
Programming with Delphi 1879
progress bars 2046
Progress Bars 2046
project directory
VCL Forms 106
projects
WebSnap 149
properties
component writing 1249, 1251
events 1351
methods 2056
setting 2148
storing properties 1254
types 1250
Properties That Contain Request Header Information 2274
Properties That Describe the Content 2275
Properties That Describe the Expected Response 2275
Properties That Describe the Web Client 2275
Properties That Identify the Purpose of the Request 2275
Properties That Identify the Target 2276
Properties, Methods, and Events 2056
property
keywords, C++ 527
Property Categories 1350
Property Declarations 1251
property editors 2148
creating 1345
property page

adding controls 1416
associating with ActiveX controls 1416
updating 1416
property page wizard
creating a new property page 1416
protected 560
keywords, C++ 560
protected methods
methods 1245
Protecting Methods 1245
protocols
Internet 2253
providers
custom events 1817
error handling 1816
fetching data 1812
updating data 1808
XML 1846
Providing a Pop-up Menu 2164
Providing an OnChange Event 1270
Providing Help for Your Component 1351
pseudovariables
lexical elements, C++ 681
public 561
keywords, C++ 561
published
keywords, C++ 528
Publishing Inherited Properties 1250
Publishing Inherited Properties (Graphic) 1227
Publishing Inherited Properties (Grid) 1271
Publishing the Pen and Brush 1228
punctuators
lexical elements, C++ 682
Punctuators 682
Punctuators Overview 681
Pure virtual function called (C++) 343
putc, putwc 1057
putch 757
putchar, putwchar 1057
putenv, _wputenv 1126
puts, _putws 1059
puttext 758
Putting a Button in a Column 1761

Q

qsort 1127
queries
datasets 1674
executing 1596
heterogenous 1676
parameters 1617
preparing 1606
setting parameters at design time 1582
setting parameters at runtime 1583
unidirectional cursors 1624
update objects 1691, 1696
updating a read-only result set 1677

R

radio buttons 2046
Radio Buttons 2046
raise 999
Raising an Exception 2020
rand 1128
RAND_MAX #define 1095
random 1128
randomize 1129
ranges
applying 1585
modifying 1601
records 1597
Rave component overview 1552
Rave Component Overview 1552
Rave Reports
creating reports 35
getting started 1555
overview 1556
Tools 81

Rave Reports: Overview 1556
Rave Visual Designer 1556
RC.EXE, the Microsoft SDK Resource Compiler 213
read 905
    write 1252
readdir, wreaddir 796
Reading a String and Writing It To a File 136
Reading and Setting Pixels 2196
Reading and Writing Events 2340
Reading and writing over socket connections 2339
Reading and Writing Over Socket Connections 2341
ReadOnly property
    FReadOnly 1330
realloc 729
Receiving Adapter Requests and Generating Responses 2318
Reconciling Update Errors 1737
records
    adding 1583
    changing 1601
    deleting 1591
    filters 1631
    finding 1630
    iterating through 1621
    marking 1599
    navigating 1603, 1634
    navigating and manipulating 1767
    posting 1606
    refreshing 1724
    specifying ranges 1610
    updating 1731
rectangle drawing 2189
Reference 155
Reference Arguments 451
reference declarations
    language structure, C++ 639
reference fields 1867
Reference/Deference Operators 658
references
    C++ language specifics 452
references, arguments
    C++ language specifics 451
references, simple
    C++ language specifics 453
Referencing 451, 452
referential integrity
    stored procedures 1570
Referential Integrity, Stored Procedures, and Triggers 1570
Refining Line Drawing 2196
Refining the Shape Drawing 1228
Refreshing Data Display 1776
Refreshing Records 1724
Refreshing the Screen 2196
Refreshing the Type Library 1467
register 561
    keywords, C++ 561
Registering a COM Object 1440
Registering Actions 1996
Registering an Active Server Object 1405
registering components
    Tool palette 1352
Registering Components 1352
Registering Components (Introduction) 1321
Registering Help System Objects 1897
Registering Multiple Properties at Once 1352
Registering Nonscalar Types 2294
Registering One Property at a Time 1353
Registering the Application Server 1535
Registering the Component 1320
Registering the Component Editor 1353
Registering the Property Editor 1354
Registering the Type Library 1467
RegisterPropertyEditor
    registering property editors 1354
Registry
    system registry 2099
Regular Expression for CodeGear Grep 170
Regular Methods (C++) 1375
reinterpret_cast

typecasting, C++ 561
reinterpret_cast (typecast Operator) 561
relational operators
operators, C++ 591
Remotable Object Example 2297
remotable objects 2297
using 2296
remote connections
brokering 1520
managing 1533
using DCOM 1537
using HTTP 1537
using SOAP 1538
using TCP/IP 1538
remote data modules
creating 1524, 1525
setting up 1536
remove, _wremove 1060
Removing Attribute Associations 1860
Removing Dependencies 1321
rename
renaming 137
VCL Forms 137
rename, _wrename 1061
Renaming a File 2130
Renaming a Table 1671
Renaming Files 137
Reorganizing Rows and Columns in Decision Grids 1792
Replacing the Picture 2197
Reporting Procedures 81
reports
Rave 81
Rave Reports 35
Representing a Dataset in HTML 2265
Representing Calculated Values 1725
Representing the Records in a Table 1832
Representing the Results of a Query 1832
Representing the Results of a Stored Procedure 1833
Requesting Data from the Source Dataset or Document 1725
requires 577
keywords, C++ 577
Reraising Exceptions 2020
Reserved Words 584
resizing
dynamic 1946
Resizing the Cells 1272
resolving
influencing generated SQL 1815
Resolving Update Errors On the Provider 1816
resource DLLs
locales 1931
Resource Errors 7
resource files
menus 1988
resources
isolating 1931
Responding to a Mouse Move 2198
Responding to a Mouse-down Action 2198
Responding to a Mouse-up Action 2199
Responding to Changes 1380
Responding to Changes Mediated by the Data Source 1776
Responding to Clicks 1996
Responding to Client Data Requests 1816
Responding to Client Requests 2249
Responding to Client Update Requests 1808
Responding to Client-generated Events 1817
Responding to data changes 1334
Responding to Data Changes 1334
Responding to Key-down Messages 1335
Responding to Mouse-down Messages 1336
Responding to Palette Changes 1380
Responding to Request Messages with Action Items 2280
Responding to socket events 2341
Responding to Socket Events 2343
Responding to System Events 1302
Responding to the Mouse 2199
Responding to User Actions at Runtime 1761
response messages

status 2285
response templates
HTML templates 2269
Restoring Default Values to a Column 1762
restrict 579
keywords, C++ 579
Restricting Field Values with Radio Controls 1748
Resume method
Suspend method 2242
Retrieving Data from Forms 1997
Retrieving Data from Modal Forms 1997
Retrieving Data from Modeless Forms 1999
Retrieving Information About a Session 1655
Retrieving Result Sets with Commands 1488
return 562
keywords, C++ 562
Returning a PChar Local Variable 2131
reusing code
techniques 2000
Reusing Code and Delegation 2076
Reusing Components and Groups of Components 2000
rewind 1062
rewinddir, wrewinddir 797
Ribbon Controls
adding commands to the ribbon 144
creating an application 143
rich edit controls
memo controls 2044
RIDL File 1468
RLINK32 163
RLINK32.DLL 204
RLINK32.DLL, the Resource Linker (C++) 204
rounded rectangles 2189
RTTI
C++ language specifics 453
rubber banding example 2199
Rubber Banding Example 2199
Running the Completed Application 2257
Runtime Packages 2220
Runtime type identification
C++ language specifics 454
Run-time Type Identification (RTTI) 453
Runtime Type Identification (RTTI) Overview 453
Runtime Type Identification And Destructors 454
rvalue References (C++0x) 495

S

S_Ixxxx #defines 1187
SafeArrays 1455
safecall
events 1413
properties 1412
Saving a Menu as a Template 2001
Saving a Picture to a File 2200
Saving a Type Library 1466
Saving and Registering Type Library Information 1465
Saving Changes 1726
Saving Data to a File or Stream 1740
Saving Decision Graph Series Settings 1792
scanf, wscanf 1064
SCHAR_xxx #defines 914
scope
C++ language specifics 430
language structure, C++ 597
objects 2070
Scope 597
Scope and Qualifiers 2070
Scope of Exception Handlers 2021
Scope Resolution Operator :: 454
scope, resolution operator
operator overloading, C++ 454
scope, rules
C++ language specifics 431
scoped enums
C++ 498
screen refreshing 2196
Screen Resolutions and Color Depths 1953
Screen Scaling and ChangeScale 2169

Screening Individual Updates 1817
Script Objects 2333
scroll bars 2156
  controls 2046
Scroll Bars 2046
scroll boxes 2047
Scroll Boxes 2047
SDI applications
  single document interface 1901
SDI Applications 1901
search
  Find methods 1594
  Goto methods 1595
  partial keys 1608
  repeating or extending 1607
  specifying current record 1612
search criteria
  index-based searches 1627
  searching for data 1628
Searching Datasets 1630
Searching for a Database Connection 1654
Searching On Partial Keys 1607, 1608
searchpath, wsearchpath 817
SelectAll 2165
Selecting a Server Type 2314
Selecting All Text 2165
Selecting Menu and Toolbar Styles 2001
Selecting Text 2165
Selecting the Current Day 1274
Selecting Web Application Module Options 2314
Selection Statements 648
Sending a Message That Does Not Execute Immediately 1303
Sending a Message Using the Windows Message Queue 1303
Sending Commands to the Server 1502
sending messages
  overview 1303
Sending Messages 1303
Sending Query or Stored Procedure Parameters 1727
Sending the Response 2285
SendMessage method
  sending messages 1303
server connections 2345
Server Connections 2345
Server Events 2343
server sockets
  events 2343
  ServerSocket component 2351
Server-side scripting in WebSnap 2332
Server-side Scripting in WebSnap 2333
service applications
  Application object 1903
  debugging 1903
  threads 1907
Service applications 1902
Service Applications 1903
Service Name Properties 1906
Service Threads 1907
services
  sockets 2338
Services and Ports 2339
Serving Client Requests 2249
sessions
  activating 1651
  getting information 1655
  in Web applications 2264
  multiple 1652
  naming 1659
  Session component 1664
set_new_handler
  C++ language specifics 435
set_new_handler function 970
set_terminate 840
set_unexpected 841
setbuf 1066
setjmp 995
setjmp.h 993
setlocale, _wsetlocale 917
setmem 968

setmode 906
Setting a Default Value for a Field 1860
Setting BDE Alias Parameters 1648
Setting Column Properties at Design Time 1763
Setting Component Properties 2148
Setting Decision Graph Template Defaults 1792
Setting Dimension State 1792
Setting Display and Edit Properties at Design Time 1860
Setting Field Component Properties at Runtime 1873
Setting Filter Options 1608
Setting Grid Options 1764
Setting IDE, Project, and Compiler Options 1900
Setting Maximum Dimensions, Summaries, and Cells 1793
Setting Options That Influence the Data Packets 1818
Setting Owned Classes' Properties 1229
Setting Pages to Require Logins 2330
Setting Persistent Field Properties and Events 1871
Setting Properties at Design Time 2147
Setting Properties at Runtime 2148
Setting Properties, Methods, and Events 1319
Setting Text Alignment 2166
Setting the Appearance of the Cool Bar 2003
Setting the Brush Bitmap Property 2201
Setting the Filter Property 1608
Setting the Initial Bitmap Size 2201
Setting the Initial Condition of a Speed Button 2004
Setting the Maximum Available Dimensions and Summaries 1793
Setting the Property Value 1354
Setting the Response Content 2285
Setting Tool Button Appearance and Initial Conditions 2004
Setting Up a Simple Dataset 1727
Setting Up Action Bands 2002
Setting Up Action Lists 2003
Setting Up Master/Detail Linked Relationships 1833
Setting Up the Remote Data Module 1536
Setting Up TSQLConnection 1834
Setting Web Item Properties 1536
setvbuf 1067
SH_xxxx #defines 997
shape drawing 2190
Shapes 2047
share.h 996
Sharing Code Among Event Handlers 2202
Sharing Frames 2005
Sharing Items Within a Project 1916
Sharing Objects in a Team Environment 1916
short 563
    keywords, C++ 563
SHRT_xxx #defines 914
Side Effects And Other Dangers 696
SIG_xxx #defines 998
signal (C RTL) 1000
signal.h 997
signed 563
    keywords, C++ 563
SIGxxxx #defines 998
Silent Exceptions 2022
Simple References 453
sin, sinl 955
sinh, sinhl 956
sizeof 563
sizeof operator
    keywords, C++ 563
Sizing Owner-draw Items 2167
Smart Pointers (C++) 2021
snprintf;snwprintf 1068
SOAP
    advantages 1548
    servers 2299
SOAP application wizard
    using 2300
sockets
    describing 2336
    errors 2342
    event handling 2343
    reading and writing 2341
    TCP/IP protocol 2337
    types of connections 2345

Sockets
advantages 1548
Software License Requirements 1954
sort order
specifying 1581
Sorting and Indexing 1728
Sorting Records with Indexes 1609
spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe, _wspawnl, _wspawnle, _wspawnlp, _wspawnlpe, _wspawnv, _wspawnve, _wspawnvp, _wspawnvpe 989
Special Types 584
Specialized Input Controls 2047
Specifying a Batch Move Mode 1682
Specifying a Connection Using DCOM 1537
Specifying a Connection Using HTTP 1537
Specifying a Connection Using SOAP 1538
Specifying a Connection Using Sockets 1538
Specifying a dBASE Index File 1672
Specifying a Palette for a Control 1380
Specifying a Provider 1729
Specifying Accelerator Keys and Keyboard Shortcuts 2005
Specifying Aggregates 1730
Specifying an Index with IndexName 1630
Specifying Application Module Components 2312
Specifying Default Database Connection Behavior 1661
Specifying Default Values 1255
Specifying Editor Attributes 1354
Specifying Menu Items 1355
Specifying No Default Value 1253
Specifying Paradox Directory Locations 1654
Specifying Property Categories 1351
Specifying Ranges 1610
Specifying the Active Menu: Menu Property 2006
Specifying the Columns 2265
Specifying the Command 1489
Specifying the Command to Execute 1835
Specifying the Components 1355
Specifying the Current Record After a Successful Search 1612
Specifying the Data to Display using TSQLDataSet 58
Specifying the Desired Server 2348
Specifying the HTML Template 2270
Specifying the New Default Property Value 1366
Specifying the Palette Page 1356
Specifying the Port 2350
Specifying the Provider using TLocalConnection or TConnectionBroker 59
Specifying the Query 1613
Specifying the Row Attributes 2265
Specifying the Stream Position and Size 2106
Specifying the Table Attributes 2265
Specifying the Table Type for Local Tables 1673
Specifying the Transaction Isolation Level 1504
Specifying What Data to Display 1836
Specifying What Fields Appear in Data Packets 1819
Specifying Whether the Connection Automatically Initiates Transactions 1489
speed button
adding to a panel 1969
assigning a glyph 1972
creating a group 1979
setting the initial condition 2004
speed buttons 2048
Speed Buttons 2048
splitter control
resizing 2048
Splitter Controls 2048
sprintf, swprintf 1068
SQL
executing commands 1502, 1829
metadata commands 1826
specifying commands 1835
SQL statements
passthrough 1699
sqrt, sqrtl 957
srand 1130
sscanf, swscanf 1069
Stack overflow (C++) 335
stackavail 729
standard events

events 1237
Starting a Drag Operation 2168
Starting and Stopping Threads 2242
Statement Keywords 584
statements
language structure, C++ 648
Statements 646, 648
statements, expression
language structure, C++ 647
statements, iteration
language structure, C++ 648
statements, jump
language structure, C++ 648
statements, labeled
language structure, C++ 648
statements, selection
language structure, C++ 648
static 564
keywords, C++ 564
Static Assertions (C++0x) 498
static members
C++ language specifics 394
Static Members 394
Static Methods 1375
static properties
C++ language specifics 410
Static Properties 410
Static Runtime Libraries 630
static_assert 565
keywords, C++ 565
static_cast
typecasting, C++ 565
static_cast (typecast Operator) 565
Status Bar 1449
status bars 2048
Status Bars 2048
stdarg.h 1003
stddef.h 1004
stderr, stdin, stdout 1070
stdio.h 1006
stdlib.h 1080
stime 1207
Storage Class Specifiers 585
storage classes
language structure, C++ 598
Storage Classes And Types 598
stored procedures
binding parameters 1669
datasets 1674
executing 1596
multiple result sets 1597
Oracle overloaded 1677
parameters 1625
preparing 1607
Storing a Custom Variant Type's Data 2107
Storing and Loading Properties 1254
Storing and Loading Unpublished Properties 1256
Storing the Internal Date 1274
stpcpy, _wstpcpy, _stpcpy 1161
strcat, _mbscat, wcscat 1162
strchr, _mbschr, wcschr 1163
strcmp, _mbscmp, wcscmp 1164
strcmpi 1165
strcoll,_stricoll, _mbscoll, _mbsicoll, wcscoll, _wcsicoll 1166
strcpy 1167
strcspn, _mbscspn, wcscspn 1168
strdup, _mbsdup, _wcsdup 1169
streams 2099
copying 107, 109
copying data 2094
position 2106
reading and writing data 2115
strerror 1170
strftime, wcsftime 1207
stricmp, _mbsicmp, _wcsicmp 1170
Stricter C++ Compiler (C++Builder 2007) 454
Stricter C++ Compiler: Binding of References and Qualifiers 455
Stricter C++ Compiler: Function Overload Resolution 458

Stricter C++ Compiler: Initialization and Conversion 459
Stricter C++ Compiler: String Literals Are Now Constants 457
Stricter C++ Compiler: Template Changes 457
String Comparison Warnings 11
String Constants 677
String Dependencies 2108
String Grids 2049
string list
    copying 107
string lists
    adding to 2134
    associated objects 2135
    copying 2135
    counting 2135
    creating 2135
    deleting from 2137
    finding strings 2138
    graphical objects 2154
    iterating 123
    iterating through 2138
    loading and saving 2138
    manipulating 2139
String to PChar Conversions 2131
string types
    conversions 2108
string.h 1139
strings
    accessing in a string list 2134
    adding and sorting 107, 109, 112, 123
    declaring and initializing 2123
    deleting 112
    enabling applications 1924
    sort list 138
    working with 2142
strlen, _mbslen, wcslen, _mbstrlen 1131
strlwr, _mbslwr, _wcslwr 1171
strncat 1172
strncmp, _mbsncmp, wcsncmp 1173
strncmpi, wcsncmpi 1174
strncoll, strnicoll, _mbsncoll, _mbsnicoll, _wcsncoll, _wcsnicoll 1175
strncpy, _mbsncpy, wcsncpy 1176
strnicmp, _mbsnicmp, _wcsnicmp 1176
strnset, _mbsnset, _wcsnset 1177
Strongly Typed Enums (C++0x) 498
strpbrk, _mbspbrk, wcspbrk 1178
strrchr, _mbsrchr, wcsrchr 1179
strrev, _mbsrev, _wcsrev 1180
strset, _mbsset, _wcsset 1181
strspn, _mbsspn, wcsspn 1182
strstr, _mbsstr, wcsstr 1182
strtod, _strtold, wcstod, _wcstold 1131
strtok, _mbstok, wcstok 1183
strtol, wcstol 1133
strtoul, wcstoul 1134
struct 566
    keywords, C++ 566
Structure Member Access 651
Structure Member Declarations 651
Structure Name Spaces 652
structures
    language structure, C++ 650, 651
Structures 649, 650
Structures And Functions 651
structures, member access
    language structure, C++ 651
structures, member declarations
    language structure, C++ 651
structures, name spaces
    language structure, C++ 652
structures, untagged
    language structure, C++ 650
structures,anonymous 698
strupr, _mbsupr, _wcsupr 1184
strxfrm, wcsxfrm 1185
Style property
    brushes 2175
Subclassing Windows Controls 1321
subcomponents

properties 1254
Supplying Parameters at Design Time 1582
Supplying Parameters at Runtime 1583
Supporting Master/detail Relationships 1539
Supporting Properties and Methods in Custom Variants 2108
Supporting State Information in Remote Data Modules 1540
swab 1135
switch 567
keywords, C++ 567
Switching Between Menus at Design Time 2006
Synchronizing Tables 1614
sys\stat.h 1186
sys\timeb.h 1190
sys\types.h 1192
system events
responding to (CLX) 1302
system, _wsystem 1136

T

tab controls 2049
Tab Controls 2049
Table Of CodeGear C++ Register Pseudovariables 681
tables
access rights 1587
batch operations 1670
creating 1587
datasets 1674
decision support components 1804
emptying 1592
exclusive access 1670
local table types 1673
master/detail realtionships 1598, 1616
renaming 1671
synchronizing 1614
Tag Editor 151
tan, tanl 958
tanh, tanhl 958
TApplication
TScreen 2007
using 2012
TCanvas
overview 2128
TCHAR Mapping 501
TComponent branch
overview 2059
TComponent Branch 2059
TControl branch
overview 2060
TControl Branch 2060
TCP
IP/distributing applications 1886
TDUMP.EXE 204
TDUMP.EXE, the File Dumping Utility 204
tell 907
template 568
Template Arguments 412
Template Body Parsing 461
Template Compiler Switches 414
Template Generation Semantics 414
templates
C++ language specifics 568
component 1975
project 1917
templates, C++ 412, 413, 434, 462
Templates 460
templates, arguments
templates, C++ 412
templates, class
templates, C++ 412
templates, compiler options
templates, C++ 414
templates, DLLs
C++ language specifics 432
templates, function
templates, C++ 433
templates, overriding
templates, C++ 434
templates, syntax checking

templates, C++ 461
Templates,in version 3.0
templates, C++ 414
Temporaries, C++ 455
TEncoding
using for Unicode 38
Tentative Definitions 602
terminate 841
Terminology and standards 2252
Terminology and Standards 2253
Testing and Debugging the Active Server Page Application 1406
Testing and Debugging the Application 1440
Testing Installed Components 1324
Testing the Component 1362
Testing Uninstalled Components 1324
text
cutting 2159
deleting 2160
in controls 2168
internationalizing 1929
selecting 2165
setting alignment 2166
Text 1929
text clearing 100, 102, 110, 113
text controls 2052
Text Controls 2052
text viewing controls
controls 2052
Text Viewing Controls 2052
textattr 759
textbackground 760
textcolor 761
textmode 762
TFileStream
Write method 136
TForm
forms 2009
The #error Control Directive 691
The #line Control Directive 693
The Content of HTTP Request Messages 2276
The Copy Constructor 421
The Data Dictionary 1680
The Decision Graph Display 1796
The delete Operator With Arrays 437
The Fundamental COM Interface, IUnknown 1387
The Fundamental Types 609
The Keyword This 394
The main() Function 626
The Marshaling Mechanism 1390
The new And delete Operators 434
The Operator new With Arrays 437
The Preprocessor 687
The Read Method 1252
The Request Method Type 2280
The Scope Resolution Operator 454
The structure of a Web Broker application 2272
The Structure of a Web Broker Application 2286
The Structure of Metadata Datasets 1837
The Structure of the Application Server 1542
The Structure of the Client Application 1543
The Target URL 2281
The Three Char Types 666
The Typeid Operator 454
The Value main() Returns 632
The Web Application Object 2288
The Web dispatcher 2287
The Web Dispatcher 2287
The Web Module 2288
The WndProc Method 1308
The Write Method 1252
this 568
keywords, C++ 568
thread functions
Execute method 2238
thread objects
New Thread Object dialog 2233
thread_local 577
keywords, C++ 577

thread-local variables
threadvar 2237
threads
avoiding simultaneous access 2226
clean-up code 2237
constructors 2234
debugging 2241
exceptions 2234
Executing 2242
multi-threaded applications 2231
priority 2242
sharing memory 2226
termination 2232
VCL objects 2235
waiting 2229, 2230
throw 569
keywords, C++ 569
Throwing An Exception (C++) 2019
time 1208
time.h 1193
time_t 1192
TIniFile
using 2131
TInterfacedObject 2072
TLIB.EXE 208
TLIB.EXE, the Library Manager 208
tm 1209
TMP_MAX #define 1024
tmpfile 1072
tmpnam, _wtmpnam 1073
toascii 784
TObject Branch 2058
toggle buttons 1971
Token Pasting With ## 695
tokens 695
lexical elements, C++ 678
Tokens 678
Tokens Overview 661
tolower, _mbctolower, towlower 785
tool buttons
adding 1971
assigning a menu to 1972
assigning images 1973
creating groups 1972, 1981
setting appearance and initial conditions 2004
tool changing with speed buttons 2184
toolbar
Type Library editor 1447
Toolbar 1447
Toolbar Controls 2050
toolbars 2050
adding hidden 1971
adding using a panel component 1969
adding using the toolbar component 1970
designing 1982
hiding and showing 1987
Tools API
creating modules 1279
editor interfaces 1295
module interfaces 1296
native IDE objects 1297
services 1293
versioning 1288
Tools for Working with the BDE 1681
TOUCH.EXE 212
toupper, _mbctoupper, towupper 786
TPersistent branch
overview 2059
TPersistent Branch 2059
TPrinter
overview 2129
track bars
controls 2051
Track Bars 2051
tracking
origin point 2204
Tracking Movement 2202
Tracking the Date 1275

Tracking the Origin Point 2204
transactions
automatic 1489
databases 1498, 1699
isolation levels 1504
multi-tiered applications 1533
Transactions 1570
transformation files
xml mapper 1843
translation units
language structure, C++ 599
Translation Units 599
Trapping Messages 1308
tree views 2051
Tree Views 2051
TRegistry
using 2099
TRegistryIniFile
using 2130
Triggering the Event 1239
TRIGRAPH 213
TRIGRAPH.EXE 213
Troubleshooting Custom Components (C++) 1356
try 569
keywords, C++ 569
try..finally statements
finally keyword 2024
TScreen
using 1985
TService
TDependency 1906
TSimpleDataSet
advantages and disadvantages 1736
using 1727
TVarData type
custom variants 2107
TWinControl
TWidgetControl branch/overview 2061
TWinControl/TWidgetControl Branch 2061
Two Kinds of Events 1239
TXMLDocument 2354
Type Categories 608
type libraries 1395, 1445
tasks 1453
Type Libraries 1395
type library
adding a CoClass 1463
adding a module 1465
adding a record or union 1464
adding an alias 1464
adding an enumeration 1464
adding an interface 1461
adding CoClass members 1463
adding properties and methods 1462
apply updates dialog 1466
creating a new 1460
deploying 1467
enabling simple data binding in an ActiveX control 1414
modifying an interface 1461
opening an existing 1460
refresh 1467
registering 1467
saving 1466
saving and registering information 1465
Type Library editor 1446
description 1446
GenTLB.exe 1468
information pages 1449, 1451
Object list pane 1448
RIDL file 1468
status bar 1449
supported types 1454
Type Library Editor 1446
Type Library Elements 1451
Type Library Wizard 1460
Type Specifier decltype (C++0x) 499
Type Specifiers 607
Type Specifiers (C++) 586

Type Trait Functions (C++0x) 463
Type trait functions and Incomplete types
C++0x features 472
Type trait functions and Intrinsic functions
C++0x features 472
Type Trait Functions Overview (C++0x) 472
type_info class 1212
typedef 569
keywords, C++ 569
typeid 570
keywords, C++ 570
typeinfo.h 1211
typename 570
keywords, C++ 570
Types of controls 2031
Types of Databases 1571
Types of Datasets 1635
Types of dbExpress Datasets 1836
Types of Events 2057
Types of Graphic Objects 2204
Types of Properties 1250
Types of socket connections 2344
Types of Socket Connections 2345
Types of Web Server Applications 2249
types, fundamental
language structure, C++ 609

U

uChild.pas
FormStyle property 94
ultoa, _ultow 1137
umask 908
Unary Operators 656
Understanding Database and Session Component Interactions 1647
Understanding datasets 1573
Understanding Datasets: Overview 1632
Understanding Invokable Interfaces 2292
Understanding Multi-tiered Database Applications 1544
Understanding Parameter Substitution in Update SQL Statements 1689
Understanding Service Protocols 2339
Understanding the component library 2053
Understanding the Component Library 2055
Understanding the Differences Between Ranges and Filters 1615
Understanding the message-handling system 1309
Understanding the Structure of a Package 2220
Undoing Changes 1730
unexpected 842
Unexpected termination during compilation [Module Seg#:offset] OR Unexpected termination during linking [Module Seg#:offset] (C++) 343
ungetc, ungetwc 1073
ungetch 764
Unicode
Enabling C++ applications 507
TCHAR mapping for C++ 501
using TEncoding 38
Unicode Character Types and Literals (C++0x) 500
Unicode for C++ 500
Unicode TCHAR mapping
C++ language specifics 502
unidirectional datasets
accessing metadata 1837
datasets 1823
defining queries 1832
defining record sets 1836
executing SQL commands 1829
opening 1830
representing tables 1832
stored procedures 1833
unidirectional DataSets
dbExpress 103
union 571
keywords, C++ 571
Union Declarations 660
unions
language structure, C++ 659
Unions 658, 659

unions, anonymous
language structure, C++ 659
unions, declarations
language structure, C++ 660
unit
creating for a component 97
installing to a package 97
unixtodos 829
unlock 909
unnamed threads
naming 2239
unsigned 571
unsigned int __array_extent(typename T, unsigned intI)
C++0x type trait functions 473
unsigned __int_array_rank(typename T)
C++0x type trait functions 473
Untagged Structures And Typedefs 650
Unwinding Exceptions (C++) 2016
update errors
reconciling 1737
update objects
accessing queries 1640
applying 1642, 1643
TUpdateSQL component 1692
UpdateObject
property pages 1417
updates
client dataset 1711
screening 1817
stored procedures 1809
updating
actions 2007
Updating a Read-only Result Set 1677
Updating Actions 2007
Updating Records 1731
Updating the Dataset 1337
Updating the Field Data Link Class 1338
Updating the Object 1417
Updating the Property Page 1416
Up-down Controls (VCL Only) 2052
URLs
hosts 2253
request targets 2281
Use __fastcall When Overriding VCL Methods 2098
Use of Storage Class Specifiers 614
User Access Rights 2331
user interface
developing 1983
User Interface Models 1902
user interfaces
internationalizing 1929
multiple records 1760
UI 1567
Users Can Override Default Handling 1240
using (declaration) 572
Using a Class to Manage Conversions 2110
Using a Client Dataset to Cache Updates 1731
Using a Client Dataset with a Provider 1732
Using a Client Dataset with File-based Data 1733
Using a Conversion Function 2113
Using a Dedicated File on Disk 1564
Using a Grid Control in Its Default State 1765
Using a Multi-Tiered Architecture 1564
Using a Picture, Graphic, or Canvas 1379
Using a Simple Dataset 1733
Using a Single Navigator for Multiple Datasets 1766
Using Action Lists 2007
Using ActionManager to Create Actions in a VCL Forms Application 102
Using ActiveX controls 1406
Using ADO datasets 1490
Using an Item 1916
Using an Object Repository Item in a Project 1917
Using an Update Component's Query Property 1640
Using an XML Broker 1544
Using an XML Document as the Client of a Provider 1845
Using an XML Document as the Source for a Provider 1846
Using and Modifying Frames 2008
Using Angle Brackets In Templates 412

Using Batch Updates 1491
Using BDE 60
Using BDE-enabled datasets 1666
Using BDE-enabled Datasets 1675
Using Brushes 2205
Using Canvas Methods to Draw Graphic Objects 2205
Using client datasets 1700
Using Client Datasets: Overview 1740
Using client sockets 2346
Using Client Sockets 2348
Using Code That the XML Data Binding Wizard Generates 2359
Using CodeGuard 48
Using COM Wizards 79
Using Command Objects 1492
Using Common Data Control Features 1777
Using Component Properties and Events in a Data Module 1913
Using Component Wrappers 1422
Using Critical Sections 2227
Using data controls 1743
Using Data Controls 1778
Using Data Explorer to Obtain Connection Information 52
Using data modules 1909
Using Data Modules 1912
Using Database Components in Data Modules 1647
Using database information in responses 2262
Using Database Information in Responses 2266
Using databases 1568
Using Databases 1572
Using Dataset Page Producers 2266
Using Datasets with Decision Support Components 1799
Using DataSnap 60
Using dbExpress 72
Using dbExpress Components 1821
Using dbExpress Datasets 1823
Using DCOM Connections 1546
Using Decision Cubes 1797
Using Decision Graphs 1795
Using Decision Grids 1801
Using Decision Pivots 1798
Using Decision Sources 1803
Using decision support components 1779
Using Decision Support Components 1800
Using Default Formatting for Numeric, Date, and Time Fields 1873
using Directive 429
Using DLLs in RAD Studio (C++) 1883
Using Drawing Tools 2206
Using Editor Interfaces 1295
Using fastmath math routines 842
Using File Streams 2133
Using Forms 1990
Using graphics in components 1376
Using Graphics in Components: Overview 1380
Using Help in a VCL Application 1898
Using IHelpSystem 1898
Using Implements for Delegation 2078
Using Implicit Connections 1505
Using Include Files 182
Using Indexes to Group Data 1734
Using Indexes to Search for Records 1615
Using Installation Programs 1954
Using Interfaces 2079
Using Interfaces Across the Hierarchy 2077
Using Interfaces in Distributed Applications 2079
Using Interfaces with Procedures 2080
Using Internally Calculated Fields in Client Datasets 1735
Using IntraWeb (VCL for the Web) 2254
Using IntraWeb Components 2258
Using IntraWeb with Web Broker and WebSnap 2260
Using Local Transactions 1698
Using Locate 1627
Using Lookup 1628
Using Maintained Aggregates 1735
Using Menu Templates 2009
Using Message Parameters 1309
Using Module Interfaces 1296
Using Multiple Remote Data Modules 1547
Using Multiple Update Objects 1641
Using Native IDE Objects 1297

Using Nested Detail Tables 1616
Using Nonscalar Types in Invokable Interfaces 2293
Using Object Fields 1862
Using Object Pascal or IDL Syntax 1455
Using Object Variables 2064
Using --p (Pascal Calling Conventions) 632
Using Page Producer Components 2270
Using Page Producers from an Action Item 2270
Using Paged Dimensions 1793
Using Parameters in Queries 1617
Using Passthrough SQL 1699
Using Pens 2206
Using Ports 2337
Using Precompiled Header Files 203
Using Predefined HTML-transparent Tag Names 2271
Using Project Templates 1917
Using Property Editors 2148
Using provider components 1805
Using Provider Components 1819
Using Query-type Datasets 1618
Using Rave Reports in RAD Studio 35
Using Reference Counting 2081
Using Remotable Objects 2296
Using Resource DLLs 1933
Using Server Constraints 1863
Using server sockets 2349
Using Server Sockets 2351
Using SOAP Connections 1548
Using socket components 2346
Using Socket Components 2348
Using Socket Connections 1548
Using Stored Procedure-type Datasets 1618
Using Streams 2099
Using Streams to Read or Write Data 2115
Using Table Producers 2267
Using Table Type Datasets 1620
Using TADODataSet 1492
Using TBatchMove 1678, 1683
Using TBatchMove (BDE) 61
Using TDatabase to Connect to Databases 1646
Using TDataSet Descendants 1619
Using TDataSetTableProducer 2266
Using TDBListBox and TDBComboBox 1778
Using Templates 462
Using TEncoding for Unicode Files 38
Using the as Operator with Interfaces 2082
Using the ASP Intrinsics 1402
Using the Automation Object Wizard 1432
Using The Backslash (\) For Line Continuation 695
Using the BDE to cache updates 1685
Using the BDE to Cache Updates 1689
Using the Borland Database Engine 1637, 1643
Using the Canvas 1381
Using the Clipboard with Graphics 2206
Using the COM Object Wizard 1431
Using The -D And -U Command-line Options 690
Using the Decision Cube Editor 1796
Using the Document Object Model 2353
Using the Eof and Bof Properties 1621
Using the Execute Method 1493
Using the First and Last Methods 1623
Using the HTML Tag Editor 151
Using the IsPropertyInCategory Function 1356
Using the Javascript Libraries 1549
Using the Main Form 2009
Using the Main VCL Thread 2235
Using the Main VCL Thread 131
Using the Menu Designer Context Menu 2010
Using the MoveBy Method 1623
Using the Multi-read Exclusive-write Synchronizer 2228
Using the Next and Prior Methods 1624
Using the object model 2062
Using the Object Model 2071
Using the Object Repository 1914, 1917
Using the Properties of the Canvas Object 2207
Using the RegisterComponents Function 1357
Using the Sender Parameter 2148
Using the Sessions Service 2332

Using the SOAP Application Wizard 2300
Using the Store-and-load Mechanism 1255
Using the TCustomVariantType Descendant 2116
Using the Type Library Editor 1453
Using the Update SQL Editor 1691
Using the VCL/RTL 2082
Using the VCL/RTL: Overview 2090
Using the Web Application Debugger 2245
Using the Web Page Editor 1549
Using the WSDL Importer 2302
Using the XML Data Binding Wizard 2358
Using Thread-local Variables 2237
Using TIniFile and TMemIniFile 2131
Using TOUCH.EXE 212
Using TQuery 1674
Using TQuery (Procedure) 63
Using TQueryTableProducer 2267
Using Transactional Data Modules 1546
Using transactions with the BDE 1698
Using Transactions with the BDE 1699
Using TRegistry 2099
Using TRegistryIniFile 2130
Using TSimpleDataSet 68
Using TSimpleObjectBroker 69
Using TSQLQuery 65
Using TSQLStoredProc (Procedure) 66
Using TSQLTable 66
Using TStoredProc 69, 1674
Using TTable 1674
Using TTable (Procedure) 70
Using TUpdateSQL to Update a Dataset 72
Using TXMLDocument 2354
Using Type-safe Generic Lists In Templates 413
Using Unidirectional Result Sets 1624
Using Update Objects to Update a Dataset 1692
Using Web Broker 2261, 2272
Using Web Connections 1550
Using Web Services 2289, 2291
Using WebSnap 2310
Using Windows Common Dialog Boxes 2013
Using XML in database applications 1840
Using XML in Database Applications 1847
Using XMLMapper 1848
utime.h 1213
Uxxxx_MAX #defines 915

V

va_arg, va_end, va_start 1003
Valid Types 1454
Value List Editors (VCL Only) 2036
values.h 1214
Variable Modifiers 614
variables, multithread
    language structure, C++ 529
variants
    custom 2096
VCL
    Architecture 36
    class library 1314
    CLX 2055
    components 36
VCL applications
    dbExpress 103
    VCL Forms 99
VCL Class Declarations 405
VCL component
    creating 97
VCL Exception Classes 2028
VCL for the Web
    IntraWeb 2255
VCL Forms
    ActiveX Active Forms 140
    ActiveX buttons 139
    buttons 107
    creating Windows application 89
    database applications 98
    decision support 91
    MDI applications 93, 94

multithreaded applications 124
SDI applications 96
SortList 138
strings 109
XML components 104
VCL Overview 36
VCL Procedures 82
vfprintf, vfwprintf 1074
vfscanf 1076
Viewing and Changing Design Options 1794
Viewing and Changing Dimension Settings 1794
Viewing and Editing Data with TDBGrid 1766
Viewing Overall Decision Graph Properties 1794
Viewing the Menu 2011
virtual 572
keywords, C++ 572
virtual base classes
C++ language specifics 403
Virtual Base Classes 403
virtual destructors
C++ language specifics 424
Virtual Destructors 424
virtual functions
C++ language specifics 449
Virtual Functions 447, 449
virtual methods
methods 1245
Virtual Methods 1374
visibility
language structure, C++ 599
Visibility 599
Vista
components available 39
visual feedback 2052
void 572
keywords, C++ 572
volatile 573
keywords, C++ 573
vprintf, vwprintf 1077
vscanf 1078
vsnprintf;vsnwprintf 1079
vsprintf, vswprintf 1079
vsscanf 910

W

W8000: Ambiguous operators need parentheses (C++) 362
W8001: Superfluous & with function (C++) 366
W8002: Restarting compile using assembly (C++) 362
W8003: Unknown assembler instruction (C++) 362
W8004: 'identifier' is assigned a value that is never used (C++) 375
W8005: Bit fields must be signed or unsigned int (C++) 302
W8006: Initializing 'identifier' with 'identifier' (C++) 366
W8007: Hexadecimal value contains more than three digits (C++) 366
W8008: Condition is always true OR W8008 Condition is always false (C++) 363
W8009: Constant is long (C++) 363
W8010: Continuation character \ found in // comment (C++) 364
W8011: Nonportable pointer comparison (C++) 374
W8012: Comparing signed and unsigned values (C++) 364
W8013: Possible use of 'identifier' before definition (C++) 375
W8014: Declaration ignored (C++) 365
W8015: Declare 'type' prior to use in prototype (C++) 371
W8016: Array size for 'delete' ignored (C++) 365
W8017: Redefinition of 'x' is not identical (C++) 368
W8018: Assigning 'type' to 'enumeration' (C++) 365
W8019: Code has no effect (C++) 370
W8020: 'identifier' is declared as both external and static (C++) 366
W8021: Handler for 'type1' hidden by previous handler for 'type2' (C++) 376
W8022: 'function1' hides virtual function 'function2' (C++) 367
W8023: Array variable 'identifier' is near (C++) 367
W8024: Base class 'class1' is also a base class of 'class2' (C++) 366
W8025: Ill-formed pragma (C++) 371
W8026: Functions with exception specifications are not expanded inline (C++) 377
W8027: Functions containing 'statement' are not expanded inline (C++) 378

W8028: Temporary used to initialize 'identifier' (C++) 373

W8029: Temporary used for parameter '???' (C++) 372

W8031: Temporary used for parameter 'parameter' OR W8029 Temporary used for parameter 'number' OR W8030 Temporary used for parameter 'parameter' in call to 'function' OR W8032 Temporary used for parameter 'number' in call to 'function' (C++) 372

W8032: Temporary used for parameter 2 in call to '???' (C++) 373

W8033: Conversion to 'type' will fail for members of virtual base 'class' (C++) 380

W8034: Maximum precision used for member pointer type 'type' (C++) 380

W8035: User-defined message (C++) 376

W8036: Non-ANSI keyword used: 'keyword' (C++) 378

W8037: Non-const function 'function' called for const object (C++) 369

W8038: constant member 'identifier' is not initialized (C++) 367

W8039: Constructor initializer list ignored (C++) 368

W8040: Function body ignored (C++) 369

W8041: Negating unsigned value (C++) 370

W8042: Initializer for object 'x' ignored (C++) 369

W8043: Macro definition ignored (C++) 368

W8044: #undef directive ignored (C++) 369

W8045: No declaration for function 'function' (C++) 374

W8046: Pragma option pop with no matching option push (C++) 377

W8047: Declaration of static function function ignored (C++) 370

W8048: Use qualified name to access member type 'identifier' (C++) 368

W8049: Use '> >' for nested templates Instead of '>>' (C++) 376

W8050: No type OBJ file present; disabling external types option. (C++) 377

W8051: Non-volatile function 'function' called for volatile object (C++) 369

W8052: Base initialization without a class name is now obsolete (C++) 362

W8053: 'ident' is obsolete (C++) 379

W8054: Style of function definition is now obsolete (C++) 371

W8055: Possible overflow in shift operation (C++) 379

W8056: Integer arithmetic overflow (C++) 376

W8057: Parameter 'parameter' is never used (C++) 370

W8058: Cannot create pre-compiled header: 'reason' (C++) 377

W8059: Structure packing size has changed (C++) 374

W8060: Possibly incorrect assignment (C++) 362

W8061: Initialization is only partially bracketed (C++) 367

W8062: Previous options and warnings not restored (C++) 379

W8063: Overloaded prefix operator 'operator' used as a postfix operator (C++) 371

W8064: Call to function with no prototype (C++) 363

W8065: Call to function 'function' with no prototype (C++) 363

W8066: Unreachable code (C++) 372

W8067: Both return and return with a value used (C++) 368

W8068: Constant out of range in comparison (C++) 365

W8069: Nonportable pointer conversion (C++) 372

W8070: Function should return a value (C++) 370

W8071: Conversion may lose significant digits (C++) 367

W8072: Suspicious pointer arithmetic (C++) 379

W8073: Undefined structure 'structure' (C++) 375

W8074: Structure passed by value (C++) 373

W8075: Suspicious pointer conversion (C++) 374

W8076: Template instance 'specifier' is already instantiated (C++) 283

W8077: Explicitly specializing an explicitly specialized class member makes no sense (C++) 283

W8078: Throw expression violates exception specification (C++) 376

W8079: Mixing pointers to different 'char' types (C++) 368

W8080: 'identifier' is declared but never used (C++) 364

W8081: Void functions may not return a value (C++) 375

W8082: Division by zero (C++) 365

W8083: Pragma pack pop with no matching pack push (C++) 282

W8084: Suggest parentheses to clarify precedence (C++) 282

W8085: Function 'function' redefined as non-inline (C++) 281

W8086: Incorrect use of #pragma alias "aliasName"="substituteName" (C++) 280

W8087: 'operator::operator==' must be publicly visible to be contained by a 'type' (C++) 283

W8089: 'type::operator<' must be publicly visible to be contained by a 'type' (C++) 283

W8090: 'type::operator<' must be publicly visible to be used with 'type' (C++) 283

W8091: 'type' argument 'specifier' passed to 'function' is a 'iterator category' iterator: 'iterator category' iterator required (C++) 283

W8092: 'type' argument 'specifier' passed to 'function' is not an iterator: 'type' iterator required (C++) 282

W8093: Incorrect use of #pragma codeseg [seg_name] ["seg_class"] [group] (C++) 280
W8094: Incorrect use of #pragma comment(<type> [,"string"]) (C++) 280
W8095: Incorrect use of #pragma message("string") (C++) 281
W8096: Incorrect use of #pragma code_seg(["seg_name"[,"seg_class"]]) (C++) 281
W8097: Not all options can be restored at this time (C++) 282
W8098: Multi-character character constant (C++) 281
W8099: Static main is not treated as an entry point (C++) 280
W8103: Path 'path' and filename 'filename' exceed maximum size of 'n' (C++) 379
W8104: Local Static with constructor dangerous for multi-threaded apps (C++) 384
W8105: %s member '%s' in class without constructors (C++) 281
W8105: Reference/Constant member 'identifier' in class without constructors (C++) 385
W8106: %s are deprecated (C++) 384
W8107: Type name expected (C++) 385
W8108: Constant in new expression requires an initializer (C++) 384
W8109: Parameter '%s' is a dependent type (C++) 385
W8110: Duplicate '%s' attribute directive ignored (C++) 384
W8112: Unresolved dependencies in expression (C++) 385
W8113: Inline function was declared with 'extern template' (C++) 384
wait 992
Waiting for a Task to Be Completed 2229
Waiting for a Thread to Finish Executing 2230
Waiting for Other Threads 2230
Waiting for Threads 132
wchar_t 573
  keywords, C++ 573
wcstol, strtol 1133
wcstombs 1137
wcstoul 1134
wctomb 1138
Weak Packaging 2223
Web Application Debugger 2245
Web Application Module Types 2323
Web Application object
  Web Broker 2288
Web Application Support
  Win32 41
Web applications
  deploying 1943, 1949
  multi-tiered 1551
web browser
  events 142
  VCL Forms 142
Web client 2275
web component
  VCL Forms 142
Web data modules
  Web applications 2324
Web Data Modules 2324
Web items
  properties 1536
Web modules 2322
  data modules 2272, 2287, 2288, 2324
  Web dispatcher 2287
Web Modules 2324
Web page editor 1549
Web page modules
  Web applications 2325
Web Page Modules 2325
Web request properties 2275
Web server applications
  adapters 2325
  applications 1885
  architecture 2286
  creating 2312, 2313, 2314, 2315, 2323, 2326
  debugging 2245
  types 2249
  Web Broker applications 1885
  WebSnap 2246
  WebSnap applications 1885
Web Services
  adding 2301
  clients 2307

exceptions 2305
importing 2302
using 2291
Web Services Overview 44
Web Services Procedure 153
WebBroker 41
WebSnap 41, 148
access rights 2331
adapter dispatcher 2317
adapter requests and responses 2318
adding login support 2328
dispatcher components 2319
dispatching action items 2320
dispatching requests and responses 2320
hello world 149
login pages 2329
login support 2330
page dispatcher 2321
projects 148
running applications 149
script objects 2333
server-side scripting 2333
sessions service 2332
tutorial 2311
WebSnap applications
hello world 149
WebSnap components 2326
WebSnap Procedures 147
What Are Events? 1234
What Goes into a Component? 1323
What Happens When an Action Fires 1984
What Is an Action? 2011
What Is an Object? 2071
What's in a Mouse Event 2207
What's in a Windows Message? 1309
When to Use Packages and DLLs 1883
When to Use TSimpleDataSet 1736
wherex 764
wherey 765
while 574
keywords, C++ 574
whitespace
lexical elements, C++ 686
Whitespace 686
Whitespace Overview 684
Why Create Properties? 1249
Why Use Packages? 2224
wide character routines 2141
Wide Character Routines 2141
Wide-character And Multi-character Constants 666
Wildcard Arguments 632
Win32 Developer's Guide 1217
Win32 Web Applications Overview 41
window 766
windowed controls
docking 2164
Windows Forms
hello world 88
projects 99
running applications 88
Windows Overview 45
Windows VCL Forms application
developing 89
Windows versions 1955
Windows XP
themes 1974
wizards
debugging 1284
implementing 1286
installing 1288
working with files and editors 1297
writing 1297
Working at the Application Level 2012
Working with ADO components 1470
Working with ADO Components 1494
Working with ADT Fields 1864
Working with Array Fields 1866
Working with Associated Datasets 1505

Working with BDE Aliases 1657
Working with BDE Handle Properties 1671
Working with components 2142
Working with Constraints 1863
Working with controls 2149
Working with Data Using a Client Dataset 1737
Working with DataSet Fields 1867
Working with Events and Event Handlers 2149
Working with Field Component Methods at Runtime 1864
Working with field components 1849
Working with Field Components: Overview 1877
Working with Files 2142
Working with Files and Editors 1297
Working with Frames 2012
Working with graphics and multimedia 2169
Working with Graphics and Multimedia: Overview 2208
Working with ini Files and the System Registry 2099
Working with Lists 2139
Working with Multimedia 2177
Working with Oracle Overloaded Stored Procedures 1677
Working with packages and components 2208
Working with Packages and Components: Overview 2211
Working with Password-protected Paradox and dBASE Tables 1662
Working with Pictures 1381
Working with Record Sets 1494
Working with Reference Fields 1867
Working with sockets 2335
Working with Sockets 2337
Working with Stored Procedure Parameters 1625
Working with String Lists 2140
Working with Strings 2142
Working with Text in Controls 2168
Working with type libraries 1441
Working with Type Libraries: Overview 1445
Working with XML Components 2354
Working with XML documents 2351
Working with XML Documents 2352
Working with XML Nodes 2355
write 911
Writing a finally Block (C++) 2023
Writing a Finally Block (Delphi) 2024
Writing a Wizard Class 1297
Writing Action Components 2013
Writing an Event Handler for the Button 2256
Writing an OnFilterRecord Event Handler 1635
Writing applications using COM 1918
Writing Applications Using COM 1918
Writing Cleanup Code 125
Writing Clean-up Code 2237
Writing Client Code Based On Type Library Definitions 1423
Writing Clients for Web Services 2307
Writing database applications 1919
Writing Database Applications 1919
Writing Exception Handlers 2024
Writing finally Blocks 2025
Writing Internet Applications 2243
Writing multi-threaded applications 2224
Writing Multi-threaded Applications 2231
Writing Reports 1568
Writing Servers that Support Web Services 2299
Writing the Implementation Method 1230
Writing the Register Procedure 1357
Writing the Thread Function 2238
Writing the Thread Function (Procedure) 134
Writing the Try Block 2026
Writing Utilities to Work with a Custom Variant Type 2117
Writing Web-based Client Applications 1551
WSDL
  generating 2306
  importing 2307
WSDLIMP.EXE, the Command Line WSDL Import Tool 214

X

XML 2352
  Data Binding wizard 2356, 2358
  database applications 1847
XML brokers 1544
XML Document

using Data Binding wizard 2359
XML documents
components 2354
converting to data packets 1841
data packets 1843
XML mapper
defining 1848
XML Nodes
working with 2355
xor
alternative representations of C++ token 574
xor, ^ 574
xor_eq
alternative representations of C++ token 577
xor_eq, ^= 577