InterBase
Sample Programs

Disclaimer

Borland International, Inc. (henceforth, Borland) reserves the right to make changes in specifications and other information contained in this publication without prior notice. The reader should, in all cases, consult Borland to determine whether or not any such changes have been made.

The terms and conditions governing the licensing of InterBase software consist solely of those set forth in the written contracts between Borland and its customers. No representation or other affirmation of fact contained in this publication including, but not limited to, statements regarding capacity, response-time performance, suitability for use, or performance of products described herein shall be deemed to be a warranty by Borland for any purpose, or give rise to any liability by Borland whatsoever.

In no event shall Borland be liable for any incidental, indirect, special, or consequential damages whatsoever (including but not limited to lost profits) arising out of or relating to this publication or the information contained in it, even if Borland has been advised, knew, or should have known of the possibility of such damages.

The software programs described in this document are confidential information and proprietary products of Borland.

Restricted Rights Legend. Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b) (3) (ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013.

Corporate Headquarters: Borland International Inc., 100 Borland Way, P. O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom.

Software Version: V3.0

Current Printing: October 1993
Documentation Version: v3.0.1

Reprint note

This documentation is a reprint of InterBase V3.0 documentation. It contains most of the information from *InterBase Previous Versions Documentation Corrections* and *InterBase Version* 3.2 *Documentation Corrections* and a new index. For information on features added since InterBase Version V3.0, consult the appropriate release notes.

Table Of Contents

Preface

This manual lists five sample InterBase programs. It presents at least one sample program for each language InterBase supports.

Who Should Read this Book

You should read this book if you want to see examples of GDML, SQL, and DSQL programs. If you are a new InterBase programmer, you should read this book in conjunction with the *Programmer's Guide*.

Using this Book

This book contains the following chapters:

Chapter 1	Lists the sample programs shown in the book and describes how to find them on the InterBase examples directory.
Chapter 2	Shows examples of GDML programs.
Chapter 3	Shows examples of SQL programs.
Chapter 4	Shows examples of DSQL programs.

Text Conventions

This book uses the following text conventions.

boldface

Indicates a command, option, statement, or utility. For example:

- Use the **commit** command to save your changes.
- Use the **sort** option to specify record return order.
- The **case_menu** statement displays a menu in the forms window.
- Use **gdef** to extract a data definition.

italic

Indicates chapter and manual titles; identifies filenames and pathnames. Also used for emphasis, or to introduce new terms. For example:

- See the introduction to SQL in the *Programmer's* Guide.
- */usr/interbase/lock_header*
- Subscripts in RSE references *must* be closed by parentheses and separated by commas.
- C permits only *zero-based* array subscript references.

`fixed width font`

Indicates user-supplied values and example code:

- `$run sys$system:iscinstall`
- `add field population_1950 long`

UPPER CASE

Indicates relation names and field names:

- Secure the RDB$SECURITY_CLASSES system relation.
- Define a missing value of *X* for the LATITUDE_COMPASS field.

Syntax Conventions

This book uses the following syntax conventions.

{braces}	Indicates an alternative item: • option::= {**vertical** | **horizontal** | **transparent**}
[brackets]	Indicates an optional item: • dbfield-expression[**not**]**missing**
`fixed width font`	Indicates user-supplied values and example code: • $run sys$system:iscinstall • `add field population_1950 long`
commalist	Indicates that the preceding word can be repeated to create an expression of one or more words, with each word pair separated by one comma and one or more spaces. For example, field_def-commalist resolves to: `field_def[,field_def[,field_def]...]`
italics	Indicates a syntax variable: **`create_blob`** *`blob-variable`* **`in`** *`dbfield-expression`*
|	Separates items in a list of choices.
⇓	Indicates that parts of a program or statement have been omitted.

InterBase Documentation

The InterBase Version 3.0 documentation set contains the following books:

Getting Started with InterBase (INT0032WW2179A) provides an overview of InterBase components and interfaces.

Database Operations (INT0032WW2178D) describes how to use InterBase utilities to maintain databases.

Data Definition Guide (INT0032WW2178F) describes how to create and modify InterBase databases.

DDL Reference (INT0032WW2178E) describes the function and syntax for each of the data definition language clauses and statements. It also lists the standard error messages for **gdef**.

DSQL Programmer's Guide (INT0032WW2179C) describes how to program with DSQL, a capability for accepting or generating SQL statements at runtime.

Forms Guide (INT0032WW2178A) describes how to create forms using the InterBase forms editor, **fred,** and how to use forms in **qli** and GDML applications.

Programmer's Guide (INT0032WW2178I) describes how to program with GDML, a relational data manipulation language, and SQL, an industry standard language.

Programmer's Reference (INT0032WW2178H) describes the function and syntax for each of the GDML and InterBase supported SQL clauses and statements. It also lists the standard error messages for **gpre**.

Qli Guide (INT0032WW2178C) describes the use of **qli**, the InterBase query language interpreter that allows you to read to and write from the database using interactive GDML or SQL statements.

Qli Reference (INT0032WW2178B) describes the function and syntax for each of the data definition, GDML, and SQL clauses and statements that you can use in **qli.**

Sample Programs (INT0032WW2178G) contains sample programs that show the use of InterBase features.

Master Index (INT0032WW2179B) contains index entries for the entire InterBase Version 3.0 documentation set.

In addition, platform-specific installation instructions are available for all supported platforms.

Chapter 1
Introduction

This chapter lists the names of the sample programs shown in this book and describes how to find the programs on the InterBase examples directory.

Overview

This book lists the following sample programs, where *extension* is a language-specific file extension:

- Two GDML programs:
  - gdml.*extension*, in all supported languages
  - array.*extension*, in C and Pascal
- One SQL program, sql.*extension*, in all supported languages
- Two DSQL programs:
  - dsql.*extension*, in all supported languages
  - full_dsql.*extension*, in C and Pascal

Finding the Sample Programs

All examples shown in this book exist on the InterBase examples directory. To find the examples, access the InterBase examples directory and look for the appropriate file name and extension.

Accessing the InterBase Examples Directory

To access the InterBase examples directory from:

- Apollo systems, see the */interbase/examples* directory.
- UNIX systems, see the */usr/interbase/examples* directory.
- VMS systems, see the *sys$sysroot:[systest.interbase]* directory.

Finding the Appropriate File Extension

To find the appropriate file extension for an example, refer to Table 1-1.

Table 1-1. File Extensions

Language	File Extension
Ada (VERDIX)	ea
Ada (Aldsys, VMS, Telesoft)	eada
BASIC	ebas
C	e
COBOL	ecob
FORTRAN (VMS)	efor
FORTRAN (Apollo)	eftn
Pascal	epas
PL/I	epli

For More Information

For more information on coding:

- A GDML program, refer to the GDML part of the *Programmer's Guide*.
- An SQL program, refer to the SQL part of the *Programmer's Guide*.
- A DSQL program, refer to the DSQL part of the *Programmer's Guide*.

Chapter 2
GDML Examples

This chapter lists a GDML program called Tourist Information in the following supported host languages:

- Ada
- BASIC
- C
- COBOL
- FORTRAN
- Pascal
- PL/1

It also lists a program called Weather Array in C and Pascal.

Overview

An overview of the two GDML example programs is presented below. For step-by-step information on writing a GDML program, refer to the GDML part of the *Programmer's Guide*.

Tourist Information

The Tourist Information program updates and prints the tourism information for New England and the Pacific Northwest. The program first prints and optionally modifies the GUIDEBOOK field for each state. Then it creates a database that is a guide to Northern U.S. seacosts.

The Tourist Information program demonstrates the following GDML features:

- Transactions
- Blob processing routines
- Error handling

This program is available online in the InterBase examples directory under the name gdml.*extension*, where *extension* is the language extension listed in Chapter 1, *Introduction*.

Weather Array

The Weather Array program stores, modifies, and reports on temperature and rainfall data for the years 1973 to 1975. It organizes this information by city and state.

The Weather Array program demonstrates the following GMDL features:

- Arrays
- Forms with dynamic menus

This program is available online in the InterBase examples directory, under the name array.*extension*, where extension is the language extension listed in Chapter 1, *Introduction*.

ADA Example

Tourist Information

This example applies to all supported implementations of Ada. For formatting purposes, quoted strings in this example sometimes wrap on multiple lines. When you code this program, be sure to put the quoted strings on a single line.

```
WITH basic_io, interbase;

PROCEDURE gdml IS

---
--- It's time to update and print the tourist information
--- we have for New England and the Pacific Northwest.
--- This program first prints and optionally modifies
--- the tourist information for each state.Then,it creates
--- a database that is a guide to Northern US seacoasts.
---

DATABASE ATLAS = FILENAME "atlas.gdb";
DATABASE GUIDE = FILENAME "coastal_guide.gdb";

state_count: integer := 8;
SUBTYPE state_t IS BASED_ON ATLAS.STATES.STATE;
TYPE state_list_t IS array
  (integer RANGE 1..state_count) OF state_t;
state_list : state_list_t :=
  ("ME", "NH", "MA", "RI", "CT", "OR", "WA", "AK");

PROCEDURE clean_up (states : state_list_t) IS

--- ************************************
---
---   c l e a n _ u p
---
--- ************************************
---
--- Functional description
---
--- At the user's choice, edit the tourism blob
--- for selected states,load a new one from a named file,
--- or accept text from standard input.
--- ************************************
```

```
   filename: string(1..40);
   choice : character;
   update_tr: interbase.transaction_handle;
   state  : state_t;
   in_length: natural;
   int_length: short_integer;

BEGIN

--- Start a named transaction for update,
--- reserving the necessary relations

update_tr := 0;
START_TRANSACTION update_tr CONCURRENCY READ_WRITE
   RESERVING ATLAS.TOURISM FOR WRITE,
   ATLAS.STATES FOR READ;

FOR i IN 1..state_count LOOP
   state := states(i);
   FOR (TRANSACTION_HANDLE update_tr) S
       IN ATLAS.STATES
       CROSS T IN ATLAS.TOURISM
       OVER STATE WITH S.STATE = state

       basic_io.put ("Here's the information on ");
       basic_io.put (s.state_name);
       basic_io.new_line;
       interbase.blob_display
           (T.GUIDEBOOK, ATLAS, update_tr, "guidebook", 9);
     basic_io.put ("Is this information up to date? (Y/N) ");
       basic_io.get (choice);
       IF (choice /= 'y') AND  (choice /= 'Y').THEN
           MODIFY T USING
               basic_io.put
               ("Enter F to update from a file, E to edit the
                 description ");
               basic_io.get (choice);
               IF (choice = 'F') OR (choice = 'f') THEN
                   basic_io.put ("Enter full name of input file
");
                   basic_io.get_line (filename, in_length);
                   int_length := short_integer (in_length);
                   interbase.blob_load (T.GUIDEBOOK, ATLAS,
                   update_tr, filename, int_length);
               ELSIF (choice = 'E') OR (choice = 'e') THEN
```

```
                interbase.blob_edit (T.GUIDEBOOK, ATLAS,
                update_tr, "guidebook", 9);
            ELSE
            basic_io.put ("Enter new information from standard
                input.  ");
            basic_io.put ("Terminate with <EOF>.");
            basic_io.new_line;
            CREATE_BLOB NEW IN T.GUIDEBOOK;
            NEW.LENGTH := NEW.SEGMENT'length(1);
            LOOP
                basic_io.get_line (NEW.SEGMENT, in_length);
                NEW.LENGTH := short_integer (in_length);
                PUT_SEGMENT NEW;
                EXIT WHEN basic_io.end_of_file;
            END LOOP;
            CLOSE_BLOB NEW;
        END IF;
    END_MODIFY;
  END IF;
END_FOR;
END LOOP;
COMMIT update_tr;
END clean_up;

PROCEDURE transfer_data (states : state_list_t) IS

---
--- ************************************
---
---   t r a n s f e r _ d a t a
---
--- ************************************
---
--- Functional description
---
--- Move the tourism information for selected
--- states to the guide_book database.
--- ************************************

trans_tr: interbase.transaction_handle;
state : state_t;

BEGIN
  trans_tr := 0;
```

```
    START_TRANSACTION trans_tr CONCURRENCY READ_WRITE
        RESERVING ATLAS.TOURISM FOR READ,
        GUIDE.TOURISM FOR WRITE;

    FOR i in 1..state_count LOOP
        state := states(i);
        FOR (TRANSACTION_HANDLE trans_tr) S IN ATLAS.STATES
            CROSS T IN ATLAS.TOURISM
            OVER STATE WITH S.STATE = state

            STORE (TRANSACTION_HANDLE trans_tr) NEW
                IN GUIDE.TOURISM USING
                NEW.STATE_NAME := s.state_name;
                NEW.STATE := s.state;
                NEW.CITY := t.city;
                NEW.ZIP := t.zip;

                --- One way to copy a blob
                CREATE_BLOB N_ADDR IN NEW.OFFICE;
                FOR O_ADDR IN T.OFFICE
                    N_ADDR.SEGMENT := O_ADDR.SEGMENT;
                    N_ADDR.LENGTH := O_ADDR.LENGTH;
                    PUT_SEGMENT N_ADDR;
                END_FOR;
                CLOSE_BLOB N_ADDR;

                --- another way to copy a blob
                interbase.blob_dump (T.GUIDEBOOK, ATLAS,
trans_tr,
                    "temp.tmp", 8);
                interbase.blob_load (NEW.GUIDEBOOK, GUIDE,
trans_tr,
                    "temp.tmp", 8);
            END_STORE;
        END_FOR;
    END LOOP;

    END transfer_data;

BEGIN --- gdml
---
--- ************************************
---
---    g d m l
---
```

```
--- ************************************
---
--- Functional description
---
--- The main routine establishes a list of
--- states from the main atlas database.It's
--- subroutines allow the user to update guidebook
--- descriptions of those states,and store them
--- in a separate database.
--- ************************************)

READY ATLAS;
clean_up (state_list);
READY  "north_coast_guide.gdb" AS GUIDE;
transfer_data (state_list);

FINISH ATLAS;
FINISH GUIDE;
END gdml;
```

BASIC Example

Tourist Information

For formatting purposes, quoted strings in this example sometimes wrap on multiple lines. When you code this program, be sure to put the quoted strings on a single line.

```
10 %TITLE "UPDATE_GUIDE"

DECLARE LONG CONSTANT STATE_COUNT = 8
DECLARE STRING STATE_LIST (STATE_COUNT)
DATA  'ME', 'NH', 'MA', 'RI', 'CT', 'OR', 'WA', 'AK', 'XX'
DECLARE LONG I

!  It's time to update and print the tourist information we
!  have for New England and the Pacific Northwest. This program
!  prints and optionally modifies the tourist information for
!  each state. Then, it creates a database that is a guide to
!  northern US seacoasts.

DATABASE ATLAS = FILENAME 'atlas.gdb'
DATABASE GUIDE = FILENAME 'coastal_guide.gdb'

! ************************************
! *
! *   M A I N
! *
! ************************************
! *
! * Functional description
! *
! * The main routine establishes a list of states from the
! *   main atlas database. Its subroutines allow the user to
! * update guidebook descriptions of those states, and store
! * them in a separate database.
! ************************************

20    MAT READ STATE_LIST
30    READY ATLAS
        READY 'north_coast_guide.gdb' AS GUIDE
40    CALL CLEAN_UP (STATE_LIST (), STATE_COUNT)
50    CALL TRANSFER_DATA (STATE_LIST (), STATE_COUNT)
60    FINISH ATLAS
```

```
        FINISH GUIDE
99      END

100     SUB CLEAN_UP (STRING STATES (), LONG STATE_COUNT)

        EXTERNAL LONG CONSTANT SS$_NORMAL
        EXTERNAL LONG FUNCTION LIB$GET_INPUT ( STRING BY DESC, &
            WORD BY VALUE, WORD BY REF)
        DECLARE STRINGFILENAME, CHOICE
        DELCLARE LONGI, UPDATE_TR, STAT
        DECLARE STRINGSTATE
        DECLARE WORDSTRING_SIZE

    DATABASE ATLAS = FILENAME 'atlas.gdb'

    ! ************************************
    ! *
    ! *    c l e a n _ u p
    ! *
    ! ************************************
    ! *
    ! * Functional description
    ! *
    ! * At the user's choice, edit the tourism blob for
    ! * selected states, load a new one from a named file, or
    ! * accept text from standard input.
    ! ************************************

    ! start a named transaction for update, reserving the
    !   necessary relations

110     UPDATE_TR = 0
120     START_TRANSACTION UPDATE_TR CONCURRENCY READ_WRITE
         RESERVING
            ATLAS.TOURISM FOR WRITE,
            ATLAS.STATES FOR READ

130     FOR I = 1 TO STATE_COUNT
135     FOR (TRANSACTION_HANDLE UPDATE_TR) S IN STATES
            CROSS T IN TOURISM OVER STATE WITH S.STATE = STATES (I)
140         PRINT "Here's the information on "; S.STATE_NAME

145         CALL BLOB_$DISPLAY (T.GUIDEBOOK, ATLAS, UPDATE_TR, &
                'GUIDEBOOK',  LEN ('GUIDEBOOK'))
150         INPUT "Is this information up to date (Y/N)"; CHOICE
160         IF (CHOICE <> 'Y') AND (CHOICE <> 'y') THEN
```

```
                MODIFY T USING
                INPUT &
      "Where should input come from (F for file / E for editor)"; &
                    CHOICE
                IF (CHOICE = 'F') OR (CHOICE = 'f') THEN
                    INPUT &
                    'What is the input file name'; &
                    FILENAME
                    CALL BLOB_$LOAD BY REF (T.GUIDEBOOK, &
                    ATLAS, UPDATE_TR, &
                    FILENAME, LEN (FILENAME))
                ELSE IF (CHOICE = 'E' OR CHOICE = 'e') THEN
                    CALL BLOB_$EDIT BY REF (T.GUIDEBOOK, &
                    ATLAS, UPDATE_TR, &
                    'GUIDEBOOK', &
                    LEN ('GUIDEBOOK'))
                ELSE
                    PRINT  &
      "Enter new information from standard input. Terminate with
       <EOF>."
                    CREATE_BLOB NEW IN T.GUIDEBOOK
                    STAT = LIB$GET_INPUT ( &
                    NEW.SEGMENT BY DESC, &
                    0 BY VALUE, &
                    STRING_SIZE BY REF)
                    WHILE STAT = SS$_NORMAL
                    NEW.LENGTH = STRING_SIZE
                    PUT_SEGMENT NEW
                    STAT = LIB$GET_INPUT ( &
                    NEW.SEGMENT BY DESC, &
                    0 BY VALUE, &
                    STRING_SIZE BY REF)
                    NDET
                    CLOSE_BLOB NEW
                END IF! if choice = e
                END IF! if choice = f
            END_MODIFY
        END IF ! if we modified the field
        END_FOR
    NDET I
190   COMMIT UPDATE_TR
199   SUBEND

200    SUB TRANSFER_DATA (STRING STATES (), LONG STATE_COUNT)
```

```
   DECLARE LONG I, TRANS_TR
   DECLARE STRING  STATE

   DATABASE ATLAS = FILENAME 'atlas.gdb'
   DATABASE GUIDE = FILENAME 'coastal_guide.gdb'

   ! ************************************
   ! *
   ! *   t r a n s f e r _ d a t a
   ! *
   ! ************************************
   ! *
   ! * Functional description
   ! *
   ! * Move the tourism information for selected
   ! * states to the guide_book database.
   ! ************************************

210    TRANS_TR = 0

220    START_TRANSACTION TRANS_TR CONCURRENCY READ_WRITE
        RESERVING ATLAS.TOURISM FOR READ,
           GUIDE.TOURISM FOR WRITE

230    FOR I = 1 TO STATE_COUNT
       STATE = STATES(I)
       FOR (TRANSACTION_HANDLE TRANS_TR) S IN ATLAS.STATES
           CROSS T IN ATLAS.TOURISM OVER STATE WITH
                S.STATE = STATE
           STORE (TRANSACTION_HANDLE TRANS_TR) NEW
                   IN GUIDE.TOURISM USING
               NEW.STATE_NAME = S.STATE_NAME
               NEW.STATE = S.STATE
               NEW.CITY = T.CITY
               NEW.ZIP = T.ZIP
       ! One way to copy a blob
               CREATE_BLOB N_ADDR IN NEW.OFFICE
               FOR O_ADDR IN T.OFFICE
                   N_ADDR.SEGMENT = O_ADDR.SEGMENT
                   N_ADDR.LENGTH = O_ADDR.LENGTH
                   PUT_SEGMENT N_ADDR
               END_FOR
               CLOSE_BLOB N_ADDR
```

```
        ! another way to copy a blob
                CALL BLOB_$DUMP BY REF (T.GUIDEBOOK, ATLAS, &
                    TRANS_TR, 'TEMP.TMP', LEN ('TEMP.TMP'))
                CALL BLOB_$LOAD BY REF (NEW.GUIDEBOOK, GUIDE, &
                    TRANS_TR, 'TEMP.TMP', LEN('TEMP.TMP'))
            END_STORE
        END_FOR
    NEXT I

240 SUBEND
```

C Examples

Tourist Information

Be sure to use the **-e** (either case) switch on **gpre** when precompiling this program.

```
/*
 * It's time to update and print the tourist information
 * we have for New England and the Pacific Northwest. This
 * program first prints and optionally modifies the tourist
 * information for each state. It then creates a database
 * that is a guide to Northern US seacoasts.
 */

DATABASE atlas = FILENAME "atlas.gdb";
DATABASE guide = FILENAME "coastal_guide.gdb";

#include <stdio.h>

static char *state_list[] =
   {"ME", "NH", "MA", "RI", "CT", "OR", "WA", "AK", 0};

#define GET_STRING(p,b)  get_string (p, b, sizeof (b))
#define FALSE 0
#define TRUE 1

main ()
{
/**************************************
 *
 *   m a i n
 *
 **************************************
 *
 * Functional description
 *
 * The main routine establishes a list of
 * states from the main atlas database. Its
 * subroutines allow the user to update guidebook
 * descriptions of those states, and store them
 * in a separate database.
 **************************************/

READY atlas;
clean_up (state_list);
```

```
READY "north_coast_guide.gdb" AS guide;
transfer_data (state_list);

FINISH atlas;
FINISH guide;
}

static clean_up (states)
char  *states[];

{
/*************************************
 *
 *   c l e a n _ u p
 *
 *************************************
 *
 * Functional description
 *
 * At the users' choice, edit the tourism blob for selected
 * states,load a new one from a named file, or accept text
 * from standard input
 *************************************/

char  filename[40], choice[4], *state;
BSTREAM*tour;
int  *update_tr;
short  c;

/* start a named transaction for update,
   reserving the necessary relations */

update_tr = 0;
START_TRANSACTION update_tr CONCURRENCY READ_WRITE
  RESERVING atlas.tourism FOR WRITE,
  atlas.states FOR READ;

for (state = *states++; state; state = *states++ )
  {
  FOR (TRANSACTION_HANDLE update_tr) s IN STATES
      CROSS t IN ATLAS.TOURISM OVER state WITH s.state = state
      {
      printf ("Here's the information on %s.\n\n",
          s.state_name);
      BLOB_display (&t.guidebook, atlas, update_tr,
```

```
          "guide_book");
      GET_STRING
          ("\nIs this information up to date? (Y/N) ", choice);
      if (*choice != 'y' && *choice != 'Y')
      MODIFY t USING
          {
          GET_STRING
  ("\nEnter F to update from a file, E to edit the
          description ",
              choice);
          if (*choice == 'F' || *choice == 'f')
              {
              GET_STRING ("Enter full path name of input file",
                  filename);
              BLOB_load (&t.guidebook, atlas, update_tr,
                  filename);
              }
          else if (*choice == 'E' || *choice == 'e')
              BLOB_edit (&t.guidebook, atlas, update_tr,
                   "guidebook");
          else
              {
              printf ("Enter new information from standard     \
                   input.  ");
           printf ("Terminate with <EOF>.\n");
           if (tour = Bopen (&t.guidebook, atlas,
                  update_tr, "w"))
                  while ((c = getchar ()) != EOF)
                  putb (c, tour);
              BLOB_close (tour);
           }
          }
      END_MODIFY;
      }
  END_FOR;
  }
)MMIT update_tr;


:atic get_string (prompt_string, response_string, length)
   char*prompt_string, *response_string;
   int length;
```

```
/**************************************
 *
 *g e t _ s t r i n g
 *
 **************************************
 *
 * Functional description
 * Write a prompt and read a string.  If the string overflows,
 * try again.  If we get an EOF, return FAILURE.
 *
 **************************************/
char  *p;
short  l, c;

while (1)
    {
    printf ("%s: ", prompt_string);
    for (p = response_string, l = length; l; --l)
   {
   c = getchar();
   if (c == EOF)
       return FALSE;
   if (c == '\n')
       {
       *p = 0;
       return TRUE;
       }
   *p++ = c;
   }
     while (getchar() != '\n')
   ;
     printf ("** Maximum field length is %d characters **\n",
       length - 1);
    }
}

static transfer_data (states)
   char *states[];

{
/**************************************
 *
 *   t r a n s f e r _ d a t a
 *
```

```
 ************************************
 *
 * Functional description
 *
 * Move the tourism information for selected
 * states to the guide_book database.
 ************************************/

int*trans_tr;
short  c;
BSTREAM  *b_in, *b_out;
char  *state;

trans_tr = 0;

START_TRANSACTION trans_tr CONCURRENCY READ_WRITE
  RESERVING atlas.tourism FOR READ,
  guide.tourism FOR WRITE;

for (state = *states++; state; state = *states++ )
  {
  FOR (TRANSACTION_HANDLE trans_tr) s IN atlas.STATES
      CROSS t IN atlas.TOURISM OVER state WITH s.state = state
      {
      STORE (TRANSACTION_HANDLE trans_tr) new IN guide.TOURISM
          USING
          strcpy (new.state_name, s.state_name);
          strcpy (new.state, s.state);
          strcpy (new.city, t.city);
        strcpy (new.zip, t.zip);

          /* One way to copy a blob */
          CREATE_BLOB n_addr IN new.office;
          FOR o_addr IN t.office
              strcpy (n_addr.SEGMENT, o_addr.SEGMENT);
              n_addr.LENGTH = o_addr.LENGTH;
              PUT_SEGMENT n_addr;
          END_FOR;
          CLOSE_BLOB n_addr;
          /* another way to copy a blob */
          b_in = Bopen (&t.guidebook, atlas, trans_tr, "r");
          b_out = Bopen (&new.guidebook, guide, trans_tr, "w");
          while ((c = getb (b_in)) != EOF)
              putb (c, b_out);
          BLOB_close (b_in);
```

```
            BLOB_close (b_out);
        END_STORE;
        }
    END_FOR;
    }
}
PREPARE trans tr
    ON_ERROR
            {
            printf ("Error preparing a multi_database
                transaction\n");
            printf ("Please manually rollback limbo transactions
                in GUIDE and ATLAS.");
            ROLLBACK trans_tr;
            return;
            }
    END_ERROR;
COMMIT trans_tr
    ON_ERROR
            {
            printf ("Error committing a prepared multi_database
                transaction\n");
            printf ("Please manually commit limbo transactions in
                GUIDE and ATLAS.");
            }
    END_ERROR;
}
```

Weather Array

Be sure to use the **-e** (either case) switch on **gpre** when precompiling this program.

```
database db = 'atlas.gdb';

#define RAIN 1
#define TEMP 2

double tot_array();
double avg_array();
double avg_month();

double rain_array [12];
double temperature_array [12];
```

```
based_on cities.city city;
based_on cities.state state;

int *trans;

main()
{
   int cont = 1;

   ready;

   start_transaction;

         /* loop at main menu until EXIT is choosen */

   while (cont) {

       case_menu "Array Demo - Pick an Operation"

           menu_entree         "Store / Modify Temperatures" :
               mod_temp();

           menu_entree         "Store / Modify Rain Fall" :
               mod_rain();

           menu_entree         "Report" :
               reports();
           menu_entree         "Exit"        :
               cont = 0;
       end_menu;

       }

   commit;

   finish;
}
/*************************************************************/
/*                                                           */
/* modify rainfall figures for a city and state              */
/* combination for 1973 - 1975                               */
/*                                                           */
```

```
/*************************************************************/
int mod_rain()
{
   int year;
   int i, j;
   int loop = 1;
   int loop2 = 0;

   trans = 0;

   START_TRANSACTION trans;

   /* go and get a city & state to operate on */

   if ( get_city_state() == -1)
       return;

   /* loop on year to modify until EXIT is choosen */

   while ( year = get_year(RAIN) ) {

       for form (TRANSACTION_HANDLE trans) f in arrays

           for (TRANSACTION_HANDLE trans) first 1 c in cities
               with c.city = city and c.state = state

               /* check and see if array had previously been  */
               /* populated. If not, set local array to all   */
               /* zeroes, else copy in values from databse    */
               /* to local array.                             */

               if (c.rain_array.NULL == GDS_$FALSE) {
                   j = year - 1973;
                   for (i=0;i<12;i++) {
                   rain_array[i] = c.rain_array[j][i];
                   }
                   }
               else {
                   for (i=0;i<12;i++) {
                   rain_array[i] = 0;
                   }
                   }
```

```
end_for;

/* assign local array values to form fields */

f.m1 = rain_array [0];
f.m2 = rain_array [1];
f.m3 = rain_array [2];
f.m4 = rain_array [3];
f.m5 = rain_array [4];
f.m6 = rain_array [5];
f.m7 = rain_array [6];
f.m8 = rain_array [7];
f.m9 = rain_array [8];
f.m10 = rain_array [9];
f.m11 = rain_array [10];
f.m12 = rain_array [11];

strcpy(f.msg_line1, "          Store / Modify Rain  \
       Fall Entry");
strcpy(f.msg_line2, "Enter Rain Fall - F1 to SAVE   \
       or F2 to ROLLBACK");
strcpy(f.state, state);
strcpy(f.city, city);
f.year = year;

loop = 1;
loop2 = 0;

while (loop) {

    display f
        displaying *
        cursor on m1
        accepting m1, m2, m3, m4, m5, m6, m7, m8,
         m9, m10, m11, m12;


    switch (f.TERMINATOR) {
        case PYXIS_$KEY_PF1:
        loop = 0;
        break;
        case PYXIS_$KEY_PF2:
        loop2 = 1;
```

```
                    loop = 0;
                    break;
                    default:
                    strcpy(f.msg_line2, "INVALID KEY - F1      \
                     to SAVE or F2 to ROLLBACK");
                    break;
                    }

                } /* while (loop) */

            /* if PF2 is pressed then don't update values in */
            /* database                                      */

            if (loop2)
                continue;

            /* update database using values in form */

            for (TRANSACTION_HANDLE trans) c in cities
                 with c.city = city and c.state = state
                modify c using
                    j = year - 1973;
                    c.rain_array[j][0] = f.m1;
                    c.rain_array[j][1] = f.m2;
                    c.rain_array[j][2] = f.m3;
                    c.rain_array[j][3] = f.m4;
                    c.rain_array[j][4] = f.m5;
                    c.rain_array[j][5] = f.m6;
                    c.rain_array[j][6] = f.m7;
                    c.rain_array[j][7] = f.m8;
                    c.rain_array[j][8] = f.m9;
                    c.rain_array[j][9] = f.m10;
                    c.rain_array[j][10] = f.m11;
                    c.rain_array[j][11] = f.m12;
                end_modify;
            end_for;

        end_form;

        }

    COMMIT trans;
```

```
  return;
}
/************************************************************/
/*                                                          */
/* modify temperature figures for a city and state          */
/* combination for 1973 - 1975                              */
/*                                                          */
/************************************************************/
int mod_temp()
{
  int year;
  int i, j;
  int loop = 1;
  int loop2 = 0;

  trans = 0;


  START_TRANSACTION trans;

  /* go and get a city & state to operate on */

  if ( get_city_state() == -1)
      return;

  /* loop on year to modify until EXIT is choosen */

  while ( year = get_year(TEMP) ) {

      for form (TRANSACTION_HANDLE trans) f in arrays

          for (TRANSACTION_HANDLE trans) first 1 c in cities
              with c.city = city and c.state = state

              /* check and see if array had previously been  */
              /* populated.If not, set local array to all    */
              /* zeroes,else copy in values from database to */
              /* local array.                                */

              if (c.temperature_array.NULL == GDS_$FALSE) {
                  j = year - 1973;
                  for (i=0;i<12;i++) {
                  temperature_array[i] =
```

```
c.temperature_array[j][i];
                    }
                    }
                else {
                    for (i=0;i<12;i++) {
                    temperature_array[i] = 0;
                    }
                    }

            end_for;

            /* assign local array values to form fields */

            f.m1 = temperature_array [0];
            f.m2 = temperature_array [1];
            f.m3 = temperature_array [2];
            f.m4 = temperature_array [3];
            f.m5 = temperature_array [4];
            f.m6 = temperature_array [5];
            f.m7 = temperature_array [6];
            f.m8 = temperature_array [7];
            f.m9 = temperature_array [8];
            f.m10 = temperature_array [9];
            f.m11 = temperature_array [10];
            f.m12 = temperature_array [11];

            strcpy(f.msg_line1, "          Store / Modify      \
                    Temperatures Entry");
            strcpy(f.msg_line2, "Enter Temperatures - F1 to    \
                    SAVE or F2 to ROLLBACK");
            strcpy(f.state, state);
            strcpy(f.city, city);
            f.year = year;

            loop = 1;
            loop2 = 0;

            while (loop) {

                display f
                    displaying *
                    cursor on m1
                    accepting m1, m2, m3, m4, m5, m6, m7, m8,
```

```
        m9, m10, m11, m12;

    switch (f.TERMINATOR) {
        case PYXIS_$KEY_PF1:
        loop = 0;
        break;
        case PYXIS_$KEY_PF2:
        loop2 = 1;
        loop = 0;
        break;
        default:
        strcpy(f.msg_line2, "INVALID KEY - F1  \
               to SAVE or F2 to ROLLBACK");
        break;
        }

    } /* while (loop) */

/* if PF2 is pressed then don't update values   */
/* in database                                  */

if (loop2)
    continue;

/* update database using values in form */

for (TRANSACTION_HANDLE trans) c in cities
    with c.city = city and c.state = state
    modify c using
        j = year - 1973;
        c.temperature_array[j][0] = f.m1;
        c.temperature_array[j][1] = f.m2;
        c.temperature_array[j][2] = f.m3;
        c.temperature_array[j][3] = f.m4;
        c.temperature_array[j][4] = f.m5;
        c.temperature_array[j][5] = f.m6;
        c.temperature_array[j][6] = f.m7;
        c.temperature_array[j][7] = f.m8;
        c.temperature_array[j][8] = f.m9;
        c.temperature_array[j][9] = f.m10;
        c.temperature_array[j][10] = f.m11;
        c.temperature_array[j][11] = f.m12;
    end_modify;
```

```
            end_for;

        end_form;

        }

    COMMIT trans;

    return;
}

/************************************************************/
/*                                                          */
/* print a report for a city and state combination that     */
/* includes all the rainfall and temperature information    */
/* for 1973 through 1975.                                   */
/*                                                          */
/************************************************************/
int reports()
{
    int year;
    int i, j;
    int loop = 1;
    int loop2 = 0;
    char *under_string  = "\t\t -----  -----  -----  -----  ----
-  -----  -----  -----  -----  -----  -----  -----  ------  -
-----\n";
    char *under_string2 = "\t\t -----  -----  -----  -----  ----
-  -----  -----  -----  -----  -----  -----  -----\n";
    char *print_string = "\t%d\t %5.2f  %5.2f  %5.2f  %5.2f  %5.2f
%5.2f  %5.2f  %5.2f  %5.2f  %5.2f  %5.2f  %5.2f  %6.2f  %6.2f\n";
    char *print_string2 = "\tAvg\t %5.2f  %5.2f  %5.2f  %5.2f
%5.2f  %5.2f  %5.2f  %5.2f  %5.2f  %5.2f  %5.2f  %5.2f\n";
    char print_str[256];

    trans = 0;

    START_TRANSACTION trans;

    /* go and get a city & state to operate on */

    get_city_state();

    for (TRANSACTION_HANDLE trans) first 1 c in cities
```

```
        with c.city = city and c.state = state

      printf("\n\n\t\t\t\t\tTemperature and RainFall       \
          Report\n\n");
      printf("\n\t\t\t\t\t\t%s, %s\n\n\n", city, state);


      printf("\t\t  Jan    Feb    Mar    Apr    May    Jun   \
   Jul    Aug    Sep    Oct    Nov    Dec    Total    Avg.\n");
      printf("\t\t----------------------------------------
-  \
-------------------------------------------------------\n\n");

      printf("Rain Fall\n");
      for (j=1973;j<1976;j++) {
         sprintf ( print_str, print_string,
            j,
            c.rain_array[j-1973][0],
            c.rain_array[j-1973][1],
            c.rain_array[j-1973][2],
            c.rain_array[j-1973][3],
            c.rain_array[j-1973][4],
            c.rain_array[j-1973][5],
            c.rain_array[j-1973][6],
            c.rain_array[j-1973][7],
            c.rain_array[j-1973][8],
            c.rain_array[j-1973][9],
            c.rain_array[j-1973][10],
            c.rain_array[j-1973][11],
            tot_array (c.rain_array, j-1973),
            avg_array (c.rain_array, j-1973) );
         printf (print_str);
         }
      printf (under_string);
      for (i=0;i<12;i++)
         rain_array[i] = avg_month (c.rain_array, i);
      sprintf ( print_str, print_string2,
            rain_array [0],
            rain_array [1],
            rain_array [2],
            rain_array [3],
            rain_array [4],
            rain_array [5],
```

```
                rain_array [6],
                rain_array [7],
                rain_array [8],
                rain_array [9],
                rain_array [10],
                rain_array [11]);
        printf (print_str);

        printf("\n");

        printf("Temperatures\n");
        for (j=1973;j<1976;j++) {
            sprintf ( print_str, print_string,
                j,
                c.temperature_array[j-1973][0],
                c.temperature_array[j-1973][1],
                c.temperature_array[j-1973][2],
                c.temperature_array[j-1973][3],
                c.temperature_array[j-1973][4],
                c.temperature_array[j-1973][5],
                c.temperature_array[j-1973][6],
                c.temperature_array[j-1973][7],
                c.temperature_array[j-1973][8],
                c.temperature_array[j-1973][9],
                c.temperature_array[j-1973][10],
                c.temperature_array[j-1973][11],
                tot_array (c.temperature_array, j-1973),
                avg_array (c.temperature_array, j-1973) );
            printf (print_str);
            }
        printf (under_string);
        for (i=0;i<12;i++)
        temperature_array[i] = avg_month (c.temperature_array,
i);
        sprintf ( print_str, print_string2,
                temperature_array [0],
                temperature_array [1],
                temperature_array [2],
                temperature_array [3],
                temperature_array [4],
                temperature_array [5],
                temperature_array [6],
                temperature_array [7],
```

```
            temperature_array [8],
            temperature_array [9],
            temperature_array [10],
            temperature_array [11] );
        printf (print_str);

        printf("\n\n");

    end_for;

    COMMIT trans;

    return;
}
/*************************************************************/
/*                                                           */
/* function to total array for one year (index) and          */
/* return that to the caller.                                */
/*                                                           */
/*************************************************************/
double tot_array (array_1, index)
double array_1 [3] [12];
int index;
{
    double accum;
    int i;

    accum = 0;

    for (i=0;i<12;i++)
        accum += array_1 [index] [i];

    return accum;
}
/*************************************************************/
/*                                                           */
/* function to total array for one year (index) and          */
/* return the average to the caller.                         */
/*                                                           */
/*************************************************************/
double avg_array (array_1, index)
double array_1 [3] [12];
int index;
```

```
{
   double avg, accum;
   int i;

   accum = 0;

   for (i=0;i<12;i++)
       accum += array_1 [index] [i];

   avg = accum / 12;

   return avg;
}
/*************************************************************/
/*                                                           */
/* function to total array for one month (index) and         */
/* return the average to the caller                          */
/*                                                           */
/*************************************************************/
double avg_month (array_1, index)
double array_1 [3] [12];
int index;
{
   double avg, accum;
   int i;

   accum = 0;

   for (i=0;i<3;i++)
       accum += array_1 [i] [index];

   avg = accum / 3;

   return avg;
}
/*************************************************************/
/*                                                           */
/* function to query the user for a city and state to        */
/* process and also to return the status of the query.       */
/*                                                           */
/*************************************************************/
int get_city_state()
{
```

```
int loop = 0;


while (loop == 0) {

    strcpy (state, "");
    strcpy (city, "");

    /* build a dynamic menu of State Codes to choose from */

        FOR_MENU S

       strcpy(S.TITLE_TEXT, "Choose a STATE - F1 to ABORT");
       S.TITLE_LENGTH = strlen(S.TITLE_TEXT);

       FOR st IN STATES SORTED BY st.STATE
           PUT_ITEM SY IN S
               strcpy(SY.ENTREE_TEXT, st.STATE);
               SY.ENTREE_LENGTH = strlen(SY.ENTREE_TEXT);
               SY.ENTREE_VALUE = 0;
           END_ITEM;
       END_FOR;

       DISPLAY S VERTICAL;

       /* If PF1 pressed then exit this routine and */
       */ return EXIT status                        */

       if (S.TERMINATOR == PYXIS_$KEY_PF1) {
           return -1;
           }

       strcpy(state, S.ENTREE_TEXT);

    END_MENU;


    /* build a dynamic menu of Cities from the State code */
    /* picked above                                       */
        FOR_MENU S

       strcpy(S.TITLE_TEXT, "Choose a CITY - F1 to ABORT");
       S.TITLE_LENGTH = strlen(S.TITLE_TEXT);
```

```
        FOR st IN CITIES WITH st.STATE = state
            SORTED BY st.CITY
             PUT_ITEM SY IN S
                 strcpy(SY.ENTREE_TEXT, st.CITY);
                 SY.ENTREE_LENGTH = strlen(SY.ENTREE_TEXT);
                 SY.ENTREE_VALUE = 0;
             END_ITEM;

             /* Update counter to indicate a city was found   */
             /* for state code                                */

             loop++;

        END_FOR;

        DISPLAY S VERTICAL;

        /* If PF1 pressed then exit this routine and return */
      /*  EXIT status                                       */

        if (S.TERMINATOR == PYXIS_$KEY_PF1) {
            return -1;
            }

        strcpy(city, S.ENTREE_TEXT);

    END_MENU;

    /* Check loop variable to see if any cities were found */
   /* to pick from. If none found, ask the user if         */
    /* they want to pick another state code and try        */
    /* again.                                              */


    switch (loop) {
        case 0:
            /* no cities found for this state */

            case_menu "No Cities for Selected State - Pick  \
                    Option"
                menu_entree        "Select Another State" :
                loop = 0;
```

```
                menu_entree         "EXIT" :
                loop = -1;
                return loop;
            end_menu;
            break;
        default:
            /* city found and everything is fine - continue */

            break;
        }

    }

  return loop;
}
/*************************************************************/
/*                                                           */
/* function to query the user for the year to modify and     */
/* return that to the caller.                                */
/*                                                           */
/*************************************************************/
int get_year(type)
int type;
{
  if (type == RAIN) {
      case_menu "Store / Modify Rain Fall - Pick Year"
          menu_entree         "1973" :
              return 1973;

          menu_entree         "1974" :
              return 1974;

          menu_entree         "1975" :
              return 1975;

          menu_entree         "EXIT" :
              return 0;
      end_menu;
      }
  else {
      case_menu "Store / Modify Temperatures - Pick Year"
          menu_entree         "1973" :
```

```
                return 1973;

            menu_entree         "1974" :
                return 1974;

            menu_entree         "1975" :
                return 1975;

            menu_entree         "EXIT" :
                return 0;
        end_menu;
        }
}
```

COBOL Example

Tourist Information

```
IDENTIFICATION DIVISION.
PROGRAM-ID. UPDATE_GUIDE.

*
* It's time to update and print the tourist information
* we have for New England and the Pacific Northwest.
* This program first prints and optionally modifies the
* tourist information for each state. It then creates a
* database that is a guide to Northern US seacoasts.
*
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
   DATABASE ATLAS = FILENAME 'ATLASGDMLB'
   DATABASE GUIDE = FILENAME 'COASTAL_GUIDEGDMLB'
01 STATE_LIST VALUE IS 'MENHMARICTORWAAK'.
   03  STATE_CODE BASED ON ATLAS.STATES.STATE OCCURS 8 TIMES.

01     FILENAME    PIC X(40).
01     OUT_STRING  PIC X(40).
01     NEW_LINE    PIC X(55).
01     CHOICE      PIC X(4).
01     UPDATE_TR   PIC S9(9) USAGE COMP.
01     TRANS_TR    PIC S9(9) USAGE COMP.
01     STATE  BASED ON ATLAS.STATES.STATE.
01     I           PIC S9(9) USAGE COMP.
01     LEN         PIC S9(4) USAGE COMP.
01     END_INPUT   PIC S9(4) USAGE COMP.
01     DONE        PIC S9(4) USAGE COMP.
01     CRLF_BIN    PIC S9(4) USAGE COMP VALUE IS 2573.
01     CRLF   REDEFINES CRLF_BIN PIC X(2).
PROCEDURE DIVISION.
MAIN_ROUTINE.

************************************
*
* m a i n
*
************************************
```

```
*
* Functional description
*
* The main routine establishes a list of
* states from the main atlas database.  Its
* subroutines allow the user to update guidebook
* descriptions of those states, and store them
* in a separate database.
*************************************

   READY ATLAS.
   PERFORM CLEAN_UP.
   READY  'NORTH_COAST_GUIDEGDMLB' AS GUIDE.
   PERFORM TRANSFER_DATA.

   FINISH ATLAS.
   FINISH GUIDE.
   STOP RUN.

CLEAN_UP.

*************************************
*
* c l e a n _ u p
*
*************************************
*
* Functional description
*
* At the users' choice, edit the tourism blob for
* selected states, load a new one from a named file,
* or accept text from standard input.
*************************************

* start a named transaction for update,
* reserving the necessary relations

   MOVE 0 TO UPDATE_TR.
   START_TRANSACTION UPDATE_TR CONCURRENCY READ_WRITE
       RESERVING ATLAS.TOURISM FOR WRITE,
       ATLAS.STATES FOR READ

   PERFORM WITH TEST AFTER VARYING I FROM 1 BY 1 UNTIL I = 8
       MOVE STATE_CODE (I) TO STATE
       FOR (TRANSACTION_HANDLE UPDATE_TR) S IN STATES
           CROSS T IN ATLAS.TOURISM OVER STATE WITH
```

```
           S.STATE = STATE
           DISPLAY " "
           DISPLAY "Here's the information on ", S.STATE_NAME
           DISPLAY " "
           MOVE 9 TO LEN
           CALL "BLOB_$DISPLAY" USING BY REFERENCE T.GUIDEBOOK,
               ATLAS, UPDATE_TR, 'GUIDEBOOK', LEN
                    DISPLAY 'Is this information up to date?
                          (Y/N) '
-              WITH NO ADVANCING
           ACCEPT CHOICE
           IF (CHOICE NOT = 'Y') AND (CHOICE NOT = 'y') THEN
               MODIFY T USING
                   DISPLAY
-      'Enter F to update from a file, E to edit the description '
-                  WITH NO ADVANCING
                   ACCEPT CHOICE
                   MOVE 0 TO DONE
                   IF (CHOICE = 'f') OR (CHOICE = 'F') THEN
                   DISPLAY
-                  'Enter full name of input file '
-                  WITH NO ADVANCING
                   ACCEPT FILENAME
                   UNSTRING FILENAME DELIMITED BY " "
                   INTO OUT_STRING COUNT IN LEN
                   CALL "BLOB_$LOAD" USING BY REFERENCE
                   T.GUIDEBOOK, ATLAS, UPDATE_TR,
                   OUT_STRING, LEN
                   MOVE 1 TO DONE
                   END-IF
                   IF (CHOICE = 'e') OR  (CHOICE = 'E') THEN
                   MOVE 9 TO LEN
                   CALL "BLOB_$EDIT" USING BY REFERENCE
                   T.GUIDEBOOK, ATLAS, UPDATE_TR, 'GUIDEBOOK',
LEN
                   MOVE 1 TO DONE
                   END-IF
                   IF DONE NOT = 1
                   DISPLAY
-              'Enter new information from standard input.  '
-                  WITH NO ADVANCING
                   DISPLAY 'Terminate with <EOF>.'
                   MOVE 0 TO END_INPUT
```

```
                    CREATE_BLOB NEW IN T.GUIDEBOOK
                    MOVE 60 TO NEW.LENGTH
                    ACCEPT NEW_LINE
                    AT END MOVE 1 TO END_INPUT
                    END-ACCEPT
                    PERFORM UNTIL END_INPUT = 1
                    STRING NEW_LINE, CRLF DELIMITED BY SIZE
                    INTO NEW.SEGMENT
                    PUT_SEGMENT NEW
                    PERFORM GET_LINE
                    END-PERFORM
                    CLOSE_BLOB NEW
                    END-IF
                END_MODIFY
            END-IF
        END_FOR
    END-PERFORM
    COMMIT (UPDATE_TR).

GET_LINE.
    ACCEPT NEW_LINE
        AT END MOVE 1 TO END_INPUT.

TRANSFER_DATA.

*************************************
*
* t r a n s f e r _ d a t a
*
*************************************
*
* Functional description
*
* Move the tourism information for selected
* states to the guide_book database.
*************************************)

    MOVE 0 TO TRANS_TR.

    START_TRANSACTION TRANS_TR CONCURRENCY READ_WRITE
        RESERVING ATLAS.TOURISM FOR READ,
        GUIDE.TOURISM FOR WRITE

    PERFORM VARYING I FROM 1 BY 1 UNTIL I = 8
        FOR (TRANSACTION_HANDLE TRANS_TR) S IN ATLAS.STATES
```

```
          CROSS T IN ATLAS.TOURISM OVER STATE WITH
          S.STATE = STATE_CODE (1)

          STORE (TRANSACTION_HANDLE TRANS_TR) NEW
              IN GUIDE.TOURISM USING
              MOVE S.STATE_NAME TO NEW.STATE_NAME
              MOVE S.STATE TO NEW.STATE
              MOVE T.CITY TO NEW.CITY
              MOVE T.ZIP TO NEW.ZIP

* One way to copy a blob
              CREATE_BLOB N_ADDR IN NEW.OFFICE;
              FOR O_ADDR IN T.OFFICE
                  MOVE O_ADDR.SEGMENT TO N_ADDR.SEGMENT
                  MOVE N_ADDR.LENGTH TO O_ADDR.LENGTH
                  PUT_SEGMENT N_ADDR;
              END_FOR
              CLOSE_BLOB N_ADDR

* another way to copy a blob
              MOVE 8 TO LEN
              CALL "BLOB_$DUMP" USING BY REFERENCE
                  T.GUIDEBOOK, ATLAS, TRANS_TR,
                  'TEMP.TMP', LEN
              CALL "BLOB_$LOAD" USING BY REFERENCE
                  NEW.GUIDEBOOK, GUIDE, TRANS_TR,
                  'TEMP.TMP',  LEN
          END_STORE
      END_FOR
  END-PERFORM
  PREPARE TRANS_TR
      ON_ERROR
          DISPLAY 'Error preparing a multi_database transaction'
          DISPLAY
-  'Please manually rollback limbo transactions in GUIDE
       and ATLAS'
          ROLLBACK TRANS_TR
      END_ERROR

  IF (TRANS_TR NOT = 0) THEN
      COMMIT TRANS_TR
          ON_ERROR
              DISPLAY
-     'Error committing a prepared multi_database transaction'
              DISPLAY
```

```
-       'Please manually commit limbo transactions in GUIDE
            and ATLAS'
            END_ERROR

    DISPLAY "End of program".
```

FORTRAN Examples

Tourist Information (Apollo FORTRAN)

```
      PROGRAM UPDATE_GUIDE

      INTEGER*2          STATE_COUNT
      PARAMETER          (STATE_COUNT = 9)
      CHARACTER*2        STATE_LIST (STATE_COUNT)

C  It's time to update and print the tourist
C  information we have for New England and the Pacific
C  Northwest. This program prints and optionally
C  modifies the tourist information for
C  each state. It then creates a database that is a
C  guide to northern US seacoasts.

      DATABASE ATLAS = FILENAME 'atlas.gdb'
      DATABASE GUIDE = FILENAME 'coastal_guide.gdb'

      DATA (STATE_LIST(I), I = 1, STATE_COUNT) /
     + 'ME', 'NH', 'MA', 'RI', 'CT', 'OR', 'WA', 'AK', 'XX'/

C ************************************
C *
C *   M A I N
C *
C ************************************
C *
C * Functional description
C *
C * The main routine establishes a list of states
C * from the main atlas database. Its subroutines
C * allow the user to update guidebook descriptions
C * of those states, and store them in a separate
C * database.
C ************************************

      READY ATLAS
      CALL CLEAN_UP (STATE_LIST)
      READY 'north_coast_guide.gdb' AS GUIDE
      CALL TRANSFER_DATA (STATE_LIST)
      FINISH ATLAS
```

```
      FINISH GUIDE
      END

      SUBROUTINE CLEAN_UP (STATES)

      CHARACTER*2       STATES (*)
      CHARACTER*40      FILENAME
      CHARACTER*4       CHOICE
      INTEGER*2         I
      INTEGER*4         UPDATE_TR
      CHARACTER*2       STATE
      INTEGER*4         STAT
      INTEGER*2         STRING_SIZE
      INTEGER*2         STATE_COUNT
      PARAMETER         (STATE_COUNT = 9)

      DATABASE ATLAS = FILENAME 'atlas.gdb'

C *************************************
C *
C *    c l e a n _ u p
C *
C *************************************
C *
C * Functional description
C *
C * At the users' choice, edit the tourism blob for
C * selected states, load a new one from a named file,
C * or accept text from standard input
C *************************************

C Start a named transaction for update, reserving the
C necessary relations.

      UPDATE_TR = 0
      START_TRANSACTION UPDATE_TR CONCURRENCY READ_WRITE
       RESERVING
          ATLAS.TOURISM FOR WRITE,
          ATLAS.STATES FOR READ

100   DO 110 I = 1, STATE_COUNT
      STATE = STATES (I)
      FOR (TRANSACTION_HANDLE UPDATE_TR) S IN STATES
          CROSS T IN ATLAS.TOURISM OVER STATE WITH
                   S.STATE = STATE
      PRINT *, 'Here''s the information on ', S.STATE_NAME
```

```
      CALL BLOB_$DISPLAY (T.GUIDEBOOK, ATLAS, UPDATE_TR,
     +    'GUIDEBOOK',      +    LEN ('GUIDEBOOK'))
      CALL LIB$GET_INPUT (CHOICE,
     +    'Is this information up to date? (Y/N) ',
     +    LEN (CHOICE))

      IF (CHOICE .NE. 'Y' .AND.  CHOICE .NE. 'y') THEN
          MODIFY T USING
               CALL LIB$GET_INPUT (CHOICE,
     + 'Enter F to update from a file, E to edit the'//
     + 'description ',
     +              LEN (CHOICE))

               IF (CHOICE .EQ. 'F' .OR. CHOICE .EQ. 'f') THEN
                    CALL LIB$GET_INPUT (FILENAME,
     +                   'Enter full name of input file ',
     +                   LEN (FILENAME))
                    STRING_SIZE = INDDE (FILENAME, ' ')
                    CALL BLOB_$LOAD (T.GUIDEBOOK, ATLAS,
                         UPDATE_TR, FILENAME, STRING_SIZE - 1)
               ELSE IF (CHOICE .EQ. 'E' .OR. CHOICE .EQ. 'e')
     + THEN
               CALL BLOB_$EDIT (T.GUIDEBOOK, ATLAS, UPDATE_TR,
     +                    'GUIDEBOOK', LEN ('GUIDEBOOK'))
               ELSE
                    PRINT *,
     +   'Enter new information from standard input.'//
     +   'Terminate with ^Z.'
                    CREATE_BLOB NEW IN T.GUIDEBOOK
                    NEW.LENGTH = LEN (NEW.SEGMENT)
150                 READ (*, '(A)', IOSTAT = STAT) NEW.SEGMENT
                    IF (STAT .NE. 0) GOTO 160
                    PUT_SEGMENT NEW
                    GOTO 150
160                 CONTINUE
                    CLOSE_BLOB NEW
                END IF
            END_MODIFY
      END IF
      END_FOR

110   CONTINUE
      COMMIT UPDATE_TR
      END
```

```
      SUBROUTINE TRANSFER_DATA (STATES)

      CHARACTER*2         STATES (*)
      INTEGER*2           I
      INTEGER*4           TRANS_TR
      CHARACTER*2         STATE
      INTEGER*2           STATE_COUNT
      PARAMETER           (STATE_COUNT = 9)

      DATABASE ATLAS = FILENAME 'atlas.gdb'
      DATABASE GUIDE = FILENAME 'coastal_guide.gdb'

C *************************************
C *
C *   t r a n s f e r _ d a t a
C *
C *************************************
C *
C * Functional description
C *
C * Move the tourism information for selected
C * states to the guide_book database.
C *************************************

      TRANS_TR = 0

      START_TRANSACTION TRANS_TR CONCURRENCY READ_WRITE
       RESERVING
        ATLAS.TOURISM FOR READ,
        GUIDE.TOURISM FOR WRITE

      DO 210 I = 1, STATE_COUNT
        STATE = STATES(I)
        FOR (TRANSACTION_HANDLE TRANS_TR) S IN ATLAS.STATES
              CROSS T IN ATLAS.TOURISM OVER STATE WITH S.STATE
                  = STATE
              STORE (TRANSACTION_HANDLE TRANS_TR) NEW
                  IN GUIDE.TOURISM USING
                    NEW.STATE_NAME = S.STATE_NAME
                    NEW.STATE = S.STATE
                    NEW.CITY = T.CITY
                    NEW.ZIP = T.ZIP

C One way to copy a blob
                   CREATE_BLOB N_ADDR IN NEW.OFFICE
                   FOR O_ADDR IN T.OFFICE
```

```
                        N_ADDR.SEGMENT = O_ADDR.SEGMENT
                        N_ADDR.LENGTH = O_ADDR.LENGTH
                        PUT_SEGMENT N_ADDR
                    END_FOR
                    CLOSE_BLOB N_ADDR

C another way to copy a blob
                    CALL BLOB_$DUMP (T.GUIDEBOOK, ATLAS,
     +                  TRANS_TR 'TEMP.TMP', LEN ('TEMP.TMP'))
                  END_STORE
        END_FOR
       END
```

Tourist Information (VAX FORTRAN)

```
       PROGRAM UPDATE_GUIDE


       INTEGER*2          STATE_COUNT
       PARAMETER          (STATE_COUNT = 9)
       CHARACTER*2        STATE_LIST (STATE_COUNT)

C  It's time to update and print the tourist information we
C  have for New England and the Pacific Northwest.
C  This program prints and optionally modifies the
C  tourist information for each state. Then it creates
C  a database that is a guide to northern us seacoasts

       DATABASE ATLAS = FILENAME 'atlas.gdb'
       DATABASE GUIDE = FILENAME 'coastal_guide.gdb'

       DATA (STATE_LIST(I), I = 1, STATE_COUNT) /
      + 'ME', 'NH', 'MA', 'RI', 'CT', 'OR', 'WA', 'AK', 'XX'/



C ************************************
C *
C *   M A I N
C *
C ************************************
C *
```

```
C * Functional description
C *
C * The main routine establishes a list of states from the
C * main atlas database.  Its subroutines allow the user
C * to update guidebook descriptions of those states, and
C * store them in a separate database.
C ************************************


      READY ATLAS
      CALL CLEAN_UP (STATE_LIST)
      READY 'north_coast_guide.gdb' AS GUIDE
      CALL TRANSFER_DATA (STATE_LIST)
      FINISH ATLAS
      FINISH GUIDE
      END



      SUBROUTINE CLEAN_UP (STATES)

      CHARACTER*2       STATES (*)
      CHARACTER*40      FILENAME
      CHARACTER*4       CHOICE
      INTEGER*2         I
      INTEGER*4         UPDATE_TR
      CHARACTER*2       STATE
      INTEGER*4         STAT
      INTEGER*2         STRING_SIZE
      INTEGER*2         STATE_COUNT
      PARAMETER         (STATE_COUNT = 9)

      DATABASE ATLAS = FILENAME 'atlas.gdb'

C ************************************
C *
C *    c l e a n _ u p
C *
C ************************************
C *
C * Functional description
C *
```

```
C * At the users' choice, edit the tourism blob for selected
C * states, load a new one from a named file, or accept text
C * from standard input
C ************************************

C start a named transaction for update, reserving the
C necessary relations


      UPDATE_TR = 0
      START_TRANSACTION UPDATE_TR CONCURRENCY READ_WRITE
       RESERVING
          ATLAS.TOURISM FOR WRITE,
          ATLAS.STATES FOR READ


100   DO 110 I = 1, STATE_COUNT
      STATE = STATES (I)
      FOR (TRANSACTION_HANDLE UPDATE_TR) S IN STATES
          CROSS T IN TOURISM OVER STATE WITH S.STATE = STATE
      PRINT *, 'Here''s the information on ', S.STATE_NAME
      CALL BLOB_$DISPLAY (T.GUIDEBOOK, ATLAS, UPDATE_TR,
      + 'GUIDEBOOK', LEN ('GUIDEBOOK'))
      CALL LIB$GET_INPUT (CHOICE,
      +    'Is this information up to date? (Y/N) ',
      +    LEN (CHOICE))

      IF (CHOICE .NE. 'Y' .AND.  CHOICE .NE. 'y') THEN
          MODIFY T USING
               CALL LIB$GET_INPUT (CHOICE,
       PRINT *,
      + 'Enter F to update from a file, E to edit the'//
      + 'description '
      +              LEN (CHOICE))

               IF (CHOICE .EQ. 'F' .OR. CHOICE .EQ. 'f') THEN
                    CALL LIB$GET_INPUT (FILENAME,
      +                   'Enter full name of input file ',
      +                   LEN (FILENAME))
                    STRING_SIZE = INDEX (FILENAME, ' ')
                    CALL BLOB_$LOAD (T.GUIDEBOOK, ATLAS,
      +                  UPDATE_TR, FILENAME, STRING_SIZE - 1)
               ELSE IF (CHOICE .EQ. 'E' .OR. CHOICE .EQ. 'e')
```

```
                    THEN
                      CALL BLOB_$EDIT (T.GUIDEBOOK, ATLAS,
     +                    UPDATE_TR, 'GUIDEBOOK', LEN
('GUIDEBOOK'))
                  ELSE
                       PRINT *,
     +    'Enter new information from standard input.'//
     +    'Terminate with ^Z.'
                       CREATE_BLOB NEW IN T.GUIDEBOOK
                       NEW.LENGTH = LEN (NEW.SEGMENT)
150                    READ (*, '(A)', IOSTAT = STAT) NEW.SEGMENT
                       IF (STAT .NE. 0) GOTO 160
                       PUT_SEGMENT NEW
                       GOTO 150
160                    CONTINUE
                       CLOSE_BLOB NEW
                   END IF
               END_MODIFY
     END IF
     END_FOR

110  CONTINUE
     COMMIT UPDATE_TR
     END



     SUBROUTINE TRANSFER_DATA (STATES)

     CHARACTER*2        STATES (*)
     INTEGER*2          I
     INTEGER*4          TRANS_TR
     CHARACTER*2        STATE
     INTEGER*2          STATE_COUNT
     PARAMETER          (STATE_COUNT = 9)

     DATABASE ATLAS = FILENAME 'atlas.gdb'
     DATABASE GUIDE = FILENAME 'coastal_guide.gdb'



     C ************************************
```

```
C *
C *   t r a n s f e r _ d a t a
C *
C ************************************
C *
C * Functional description
C *
C * Move the tourism information for selected
C * states to the guide_book database.
C ************************************

      TRANS_TR = 0

      START_TRANSACTION TRANS_TR CONCURRENCY READ_WRITE
       RESERVING
        ATLAS.TOURISM FOR READ,
        GUIDE.TOURISM FOR WRITE

      DO 210 I = 1, STATE_COUNT
        STATE = STATES(I)
        FOR (TRANSACTION_HANDLE TRANS_TR) S IN ATLAS.STATES
              CROSS T IN ATLAS.TOURISM OVER STATE
                WITH S.STATE = STATE
              STORE (TRANSACTION_HANDLE TRANS_TR) NEW
                IN GUIDE.TOURISM USING
                    NEW.STATE = S.STATE
                    NEW.STATE_NAME = S.STATE_NAME
                    NEW.CITY = T.CITY
                    NEW.ZIP = T.ZIP

C One way to copy a blob
                  CREATE_BLOB N_ADDR IN NEW.OFFICE
                  FOR O_ADDR IN T.OFFICE
                       N_ADDR.SEGMENT = O_ADDR.SEGMENT
                       N_ADDR.LENGTH = O_ADDR.LENGTH
                       PUT_SEGMENT N_ADDR
                  END_FOR
                  CLOSE_BLOB N_ADDR

C another way to copy a blob
                  CALL BLOB_$DUMP (T.GUIDEBOOK, ATLAS,
+                      TRANS_TR, 'TEMP.TMP', LEN ('TEMP.TMP'))
                  CALL BLOB_$LOAD (NEW.GUIDEBOOK, GUIDE,
```

```
     +                          TRANS_TR, 'TEMP.TMP', LEN ('TEMP.TMP'))
                    END_STORE
              END_FOR
            END
```

Pascal Examples

Tourist Information (Apollo Pascal)

For formatting purposes, quoted strings sometimes wrap on multiple lines in this example. When you code this program, be sure to put all quoted strings on a single line.

```
program update_guide (input, output);

 (*
 * It's time to update and print the tourist information
 * we have for New England and the Pacific Northwest. This
 * program first prints and optionally modifies the tourist
 * information for each state. It then creates a database
 * that is a guide to Northern US seacoasts.
 *)

database atlas = filename 'atlas.gdb';
database guide = filename 'coastal_guide.gdb';

const
   state_count = 8;
type
   state_t= based_on atlas.states.state;
   state_list_t = array [1..state_count] of state_t;

var
   state_list: state_list_t :=
   ['ME', 'NH', 'MA', 'RI', 'CT', 'OR', 'WA', 'AK'];

procedure clean_up (states : state_list_t);

(************************************
 *
 *   c l e a n _ u p
 *
 ************************************
 *
 * Functional description
 *
 * At the users' choice, edit the tourism blob for
 * selected states, load a new one from a named file, or
 * accept text from standard input
 ************************************)
```

```
var
  filename: array [1..40] of char;
  choice : array [1..4] of char;
  i      : integer;
  update_tr: gds_$handle;
  state  : state_t;

begin

(* start a named transaction for update,
   reserving the necessary relations *)

update_tr := nil;
start_transaction update_tr concurrency read_write
  reserving atlas.tourism for write,
  atlas.states for read;

for i := 1 to state_count do
begin
  state := states[i];
  for (transaction_handle update_tr) s in states
      cross t in atlas.tourism over state with s.state = state
  begin
      writeln ('Here''s the information on ', s.state_name);
      blob_$display (t.guidebook, atlas, update_tr,
          'guide_book', sizeof ('guidebook'));
         write
             ('Is this information up to date? (Y/N) ');
      readln (choice);
      if (choice[1] <> 'y') and  (choice[1] <> 'Y') then
      modify t using
      begin
          write ('Enter F to update from a file,');
          write (' E to edit the description ');
          readln (choice);
          if (choice[1] = 'F') or (choice[1] = 'f') then
          begin
              write ('Enter full name of input file ');
              readln (filename);
              blob_$load (t.guidebook, atlas, update_tr,
                   filename, sizeof (filename));
          end
          else if (choice[1] = 'E') or  (choice[1] = 'e') then
              blob_$edit (t.guidebook, atlas, update_tr,
```

```
                'guidebook', sizeof ('guidebook'))
        else
        begin
            write
             ('Enter new information from standard input.  ');
         writeln ('Terminate with <EOF>.');
            create_blob new in t.guidebook;
            new.length := sizeof (new.segment);
            repeat
            begin
                readln (input, new.segment);
                put_segment new;
            end;
            until eof (input);
            reset (input);
            close_blob new;
        end;
      end;
      end_modify;
  end;
  end_for;
end;
commit update_tr;
end;

procedure transfer_data (states : state_list_t);

(************************************
 *
 *   t r a n s f e r _ d a t a
 *
 ************************************
 *
 * Functional description
 *
 * Move the tourism information for selected
 * states to the guide_book database.
 ************************************)

var
  trans_tr: gds_$handle;
  state   : state_t;
  i       : integer;
```

```
begin
trans_tr := nil;

start_transaction trans_tr concurrency read_write
  reserving atlas.tourism for read,
  guide.tourism for write;

for i := 1 to state_count do
begin
  state := states[i];
  for (transaction_handle trans_tr) s in atlas.states
      cross t in atlas.tourism over state with s.state = state
      begin
      store (transaction_handle trans_tr) new in guide.tourism
          using new.state_name := s.state_name;
          new.state := s.state;
          new.city := t.city;
        new.zip := t.zip;

          (* One way to copy a blob *)
          create_blob n_addr in new.office;
          for o_addr in t.office
              n_addr.segment := o_addr.segment;
              n_addr.length := o_addr.length;
              put_segment n_addr;
          end_for;
          close_blob n_addr;

          (* another way to copy a blob *)
          blob_$dump (t.guidebook, atlas, trans_tr,
              'temp.tmp',
              sizeof ('temp.tmp'));
          blob_$load (new.guidebook, guide, trans_tr,
              'temp.tmp',
              sizeof ('temp.tmp'));
      end_store;
      end;
  end_for;
  end;

begin
(************************************
 *
 *   m a i n
 *
```

```
 ************************************
 *
 * Functional description
 *
 * The main routine establishes a list of
 * states from the main atlas database.  Its
 * subroutines allow the user to update guidebook
 * descriptions of those states and store them
 * in a separate database.
 ************************************)

ready atlas;
clean_up (state_list);
ready  'north_coast_guide.gdb' as guide;
transfer_data (state_list);

finish atlas;
finish guide;
end.
```

```
 *
 * Functional description
 *
 * The main routine establishes a list of
 * states from the main atlas database.  Its
 * subroutines allow the user to update guidebook
 * descriptions of those states, and store them
 * in a separate database.
 ************************************)

ready atlas;
clean_up (state_list);
ready  'north_coast_guide.gdb' as guide;
transfer_data (state_list);

finish atlas;
finish guide;
end.
```

Tourist Information (VAX Pascal)

For formatting purposes, quoted strings sometimes wrap on multiple lines in this example. When you code this program, be sure to put all quoted strings on a single line.

```
program update_guide (input, output);

(*
* It's time to update and print the tourist information we
* have for New England and the Pacific Northwest. This
* This program first prints and optionally modifies the
* tourist information for each state. It then creates
* a database that is a guide to Northern US seacoasts.
*)

database atlas = filename 'atlas.gdb';
database guide = filename 'coastal_guide.gdb';

const
  state_count = 8;
type
  state_t = based_on atlas.states.state;
  state_list_t = array [1..state_count] of state_t;

var
  state_list: state_list_t :=
  ('ME  ', 'NH  ', 'MA  ', 'RI  ', 'CT  ', 'OR  ', 'WA  ',
   'AK  ');



procedure clean_up (states : state_list_t);

(*************************************
 *
 *   c l e a n _ u p
 *
 *************************************
 *
 * Functional description
 *
 * At the users' choice, edit the tourism blob for selected
 * states load a new one from a named file, or accept text
 * from standard input
 *************************************)

var
  filename: packed array [1..40] of char;
```

```
  choice : packed array [1..4] of char;
  i      : integer;
  update_tr: gds_$handle;
  state  : state_t;

begin

(* start a named transaction for update, reserving the
   necessary relations *)

update_tr := nil;
start_transaction update_tr concurrency read_write
  reserving atlas.tourism for write,
  atlas.states for read;


for i := 1 to state_count do
begin
  state := states[i];
  for (transaction_handle update_tr) s in atlas.states
      cross t in atlas.tourism over state with s.state = state
  begin
      writeln ('Here''s the information on ', s.state_name);
      blob_$display (t.guidebook, atlas, update_tr,
        'guide_book',
         %REF length ('guidebook'));
             write ('Is this information up to date? (Y/N) ');
      readln (choice);
      if (choice[1] <> 'y') and  (choice[1] <> 'Y') then
      modify t using
      begin
          write ('Enter F to update from a file, E to edit the
                 description ');
          readln (choice);
          if (choice[1] = 'F') or (choice[1] = 'f') then
          begin
              write ('Enter full name of input file ');
              readln (filename);
              blob_$load (t.guidebook, atlas, update_tr,
                  filename,
                  %REF length (filename));
          end
          else if (choice[1] = 'E') or  (choice[1] = 'e') then
```

```
                blob_$edit (t.guidebook, atlas, update_tr,
              'guidebook',
                    %REF length ('guidebook'))
            else
            begin
            write ('Enter new information from standard input.
');
             writeln ('Terminate with <EOF>.');
                create_blob new in t.guidebook;
                new.length := length (new.segment);
                repeat
                begin
                    readln (input, new.segment);
                    put_segment new;
                end;
                until eof (input);
                reset (input);
                close_blob new;
            end;
        end;
        end_modify;
    end;
    end_for;
end;
commit update_tr;
end;



procedure transfer_data (states : state_list_t);


(**************************************
 *
 *   t r a n s f e r _ d a t a
 *
 **************************************
 *
 * Functional description
 *
 * Move the tourism information for selected
 * states to the guide_book database.
```

```
 ************************************)

var
  trans_tr: gds_$handle;
  state   : state_t;
  i       : integer;

begin
trans_tr := nil;

start_transaction trans_tr concurrency read_write
  reserving atlas.tourism for read,
  guide.tourism for write;


for i := 1 to state_count do
begin
  state := states[i];
  for (transaction_handle trans_tr) s in atlas.states
      cross t in atlas.tourism over state with s.state = state
      begin
      store (transaction_handle trans_tr) new in guide.tourism
     using
         new.state := s.state;
         new.state_name := s.state_name;
         new.city := t.city;
        new.zip := t.zip;

         (* One way to copy a blob *)
         create_blob n_addr in new.office;
         for o_addr in t.office
             n_addr.segment := o_addr.segment;
             n_addr.length := o_addr.length;
             put_segment n_addr;
         end_for;
         close_blob n_addr;

         (* another way to copy a blob *)
         blob_$dump (t.guidebook, atlas, trans_tr, 'temp.tmp',
                 %REF length ('temp.tmp'));
         blob_$load (new.guidebook, guide, trans_tr,
                 'temp.tmp',
                 %REF length ('temp.tmp'));
```

```
        end_store;
        end;
    end_for;
    end;

begin
(************************************
 *
 *   m a i n
 *
 ************************************
 *
 * Functional description
 *
 * The main routine establishes a list of
 * states from the main atlas database. Its
 * subroutines allow the user to update guidebook
 * descriptions of those states, and store them
 * in a separate database.
 ************************************)

ready atlas;
clean_up (state_list);
ready  'north_coast_guide.gdb' as guide;
transfer_data (state_list);

finish atlas;
finish guide;
end.
```

Weather Array (Apollo Pascal)

For formatting purposes, quoted strings sometimes wrap on multiple lines in this example. When you code this program, be sure to put all quoted strings on a single line.

```
program array_demo (input, output);

database db = 'atlas.gdb';

const
   RAIN = 1;
        TEMP = 2;
```

```
type
  state_t = based_on cities.state;
  city_t = based_on cities.city;

  mult_array = based_on cities.rain_array;
var
  rain_array : array [1..12] of double;
  temperature_array : array [1..12] of double;

  state : state_t;
  city  : city_t;
  year  : integer;

  trans : gds_$handle;

  cont : integer;


(*************************************************************)
(*                                                           *)
(* function to query the user for a city and state to        *)
(* process and also to return the status of the query.       *)
(*                                                           *)
(*************************************************************)
function get_city_state : integer;
var
  i : integer;
  loop : integer;
begin
  loop := 0;

  while loop = 0 do
  begin

      state := ' ';
      city  := ' ';

      (* build a dynamic menu of State Codes to choose from *)

          FOR_MENU S

         S.TITLE_TEXT := 'Choose a STATE - F1 to ABORT';
```

```
        S.TITLE_LENGTH := 30;

        FOR st IN STATES SORTED BY st.STATE
            PUT_ITEM SY IN S
                for i := 1 to 4 do
                SY.ENTREE_TEXT[i] := st.STATE[i];

                SY.ENTREE_LENGTH := sizeof(st.STATE);
                SY.ENTREE_VALUE := 0;
            END_ITEM;
        END_FOR;

        DISPLAY S VERTICAL;

        (* If PF1 pressed then exit this routine and return
           EXIT status *)

        if S.TERMINATOR = PYXIS_$KEY_PF1 then
        begin
            get_city_state := -1;
            return;
        end;

        for i := 1 to 4 do
            state[i] := S.ENTREE_TEXT[i];

    END_MENU;

    (* Build a dynamic menu of Cities from the State code  *)
    (* picked above.                                       *)

        FOR_MENU S

        S.TITLE_TEXT := 'Choose a CITY - F1 to ABORT';
        S.TITLE_LENGTH := 30;

        FOR st IN CITIES WITH st.STATE = state
            SORTED BY st.CITY
            PUT_ITEM SY IN S
                for i := 1 to 25 do
                SY.ENTREE_TEXT[i] := st.CITY[i];
                SY.ENTREE_LENGTH := sizeof(st.CITY);
                SY.ENTREE_VALUE := 0;
```

```
        END_ITEM;

        (* Update counter to indicate a city was found  *)
        (*  for state code. *)

        loop := loop + 1;

    END_FOR;

    DISPLAY S VERTICAL;

    (* If PF1 pressed then exit this routine and return *)
    (* EXIT status.                                     *)

    if S.TERMINATOR = PYXIS_$KEY_PF1 then
    begin
        get_city_state := -1;
        return;
    end;

    for i := 1 to 25 do
        city[i] := S.ENTREE_TEXT[i];

END_MENU;

(* Check loop variable to see if any cities were found *)
(*  to pick from. If none was found ask users if they   *)
(* want to pick another state code and try again.       *)

if loop = 0 then
begin
    (* no cities found for this state *)

    case_menu "No Cities for Selected State - Pick Option"
        menu_entree        "Select Another State" :
            get_city_state := 0;

        menu_entree        "EXIT" :
            begin
            get_city_state := -1;
            return;
            end;
    end_menu;
end;
```

```
    end; (* end of while loop = 0 *)

    get_city_state := loop;

end;
(*************************************************************)
(*                                                           *)
(* function to query the user for the year to modify and     *)
(* return that to the caller.                                *)
(*                                                           *)
(*************************************************************)
procedure get_year( rain_or_temp : integer);
begin
  if rain_or_temp = RAIN then
  begin
      case_menu "Store / Modify Rain Fall - Pick Year"
          menu_entree         "1973" :
              year := 1973;

          menu_entree         "1974" :
              year := 1974;

          menu_entree         "1975" :
              year := 1975;

          menu_entree         "EXIT" :
              year := 0;
      end_menu;
  end
  else
  begin
      case_menu "Store / Modify Temperatures - Pick Year"
          menu_entree         "1973" :
              year := 1973;

          menu_entree         "1974" :
              year := 1974;

          menu_entree         "1975" :
              year := 1975;

          menu_entree         "EXIT" :
```

```
            year := 0;
        end_menu;
    end;

end;
(*************************************************************)
(*                                                           *)
(* modify rain fall figures for a city and state             *)
(* combination for 1973 - 1975                               *)
(*                                                           *)
(*************************************************************)
procedure mod_rain;
var
    status,
    i,
    j,
    loop,
    loop2 : integer;

begin
    loop := 1;
    loop2 := 0;

    trans := nil;

    START_TRANSACTION trans;

    (* go and get a city & state to operate on *)

    status := get_city_state;

    if status = -1 then
    begin
        return;
    end;

    (* loop on year to modify until EXIT is choosen *)

    get_year (RAIN);
    while year <> 0 do
    begin

        for form (TRANSACTION_HANDLE trans) f in arrays
```

```
for (TRANSACTION_HANDLE trans) first 1 c in cities
     with c.city = city and c.state = state

    (* check and see if array had previously been
         populated *)
    (* if not, set local array to all zero's     *)
    (*else copy in values from database to local
      array     *)

    if c.rain_array.NULL = GDS_$FALSE then
    begin
        for i := 1 to 12 do
        begin
        rain_array[i] := c.rain_array[year,i];
        end;
    end
    else
    begin
        for i := 1 to 12 do
        begin
        rain_array[i] := 0;
        end;
    end;

end_for;

(* assign local array values to form fields *)

f.m1 := rain_array [1];
f.m2 := rain_array [2];
f.m3 := rain_array [3];
f.m4 := rain_array [4];
f.m5 := rain_array [5];
f.m6 := rain_array [6];
f.m7 := rain_array [7];
f.m8 := rain_array [8];
f.m9 := rain_array [9];
f.m10 := rain_array [10];
f.m11 := rain_array [11];
f.m12 := rain_array [12];

f.msg_line1 :=
```

```
        '          Store / Modify Rain Fall Entry';
    f.msg_line2 :=
        'Enter Rain Fall - F1 to SAVE or F2 to ROLLBACK';
    f.state := state;
    f.city := city;
    f.year := year;

    loop := 1;
    loop2 := 0;

    while loop = 1 do
    begin

        display f
            displaying *
            cursor on m1
            accepting m1, m2, m3, m4, m5, m6, m7, m8, m9,
m10, m11, m12;


        case f.TERMINATOR of
            PYXIS_$KEY_PF1:
            loop := 0;
            PYXIS_$KEY_PF2:
            begin
            loop2 := 1;
            loop := 0;
            end;
            OTHERWISE
            f.msg_line2 :=
            'INVALID KEY - F1 to SAVE o F2 to ROLLBACK';
        end;
    end;

    (* if PF2 isn't pressed then update values
       in database *)

    if loop2 <> 1 then
    begin
        (* update database using values in form *)

        for (TRANSACTION_HANDLE trans) c in cities
             with c.city = city and c.state = state
```

```
                    modify c using
                    c.rain_array[year,1] := f.m1;
                    c.rain_array[year,2] := f.m2;
                    c.rain_array[year,3] := f.m3;
                    c.rain_array[year,4] := f.m4;
                    c.rain_array[year,5] := f.m5;
                    c.rain_array[year,6] := f.m6;
                    c.rain_array[year,7] := f.m7;
                    c.rain_array[year,8] := f.m8;
                    c.rain_array[year,9] := f.m9;
                    c.rain_array[year,10] := f.m10;
                    c.rain_array[year,11] := f.m11;
                    c.rain_array[year,12] := f.m12;
                    end_modify;
                end_for;
            end;

        end_form;

        get_year (RAIN);
    end;

    COMMIT trans;

end;
(*************************************************************)
(*                                                           *)
(* modify temperature figures for a city and state           *)
(* combination for 1973 - 1975                               *)
(*                                                           *)
(*************************************************************)
procedure mod_temp;
var
    status,
    i,
    j,
    loop,
    loop2 : integer;

begin
    loop := 1;
    loop2 := 0;
```

```
trans := nil;

START_TRANSACTION trans;

(* go and get a city & state to operate on *)

status := get_city_state;

if status = -1 then
begin
    return;
end;

(* loop on year to modify until EXIT is choosen *)

get_year (TEMP);
while year <> 0 do
begin

    for form (TRANSACTION_HANDLE trans) f in arrays

        for (TRANSACTION_HANDLE trans) first 1 c in cities
             with c.city = city and c.state = state

            (* check and see if array had previously been   *)
            (* populated. If not, set local array to all    *)
            (* zero's, else copy in values from database to *)
            (  local array                                  *)

            if c.temperature_array.NULL = GDS_$FALSE then
            begin
                for i := 1 to 12 do
                begin
                temperature_array[i] :=
                    c.temperature_array[year,i];
                end;
            end
            else
            begin
                for i := 1 to 12 do
                begin
                temperature_array[i] := 0;
                end;
```

```
            end;

        end_for;

        (* assign local array values to form fields *)

        f.m1 := temperature_array [1];
        f.m2 := temperature_array [2];
        f.m3 := temperature_array [3];
        f.m4 := temperature_array [4];
        f.m5 := temperature_array [5];
        f.m6 := temperature_array [6];
        f.m7 := temperature_array [7];
        f.m8 := temperature_array [8];
        f.m9 := temperature_array [9];
        f.m10 := temperature_array [10];
        f.m11 := temperature_array [11];
        f.m12 := temperature_array [12];

        f.msg_line1 :=
            '          Store / Modify Temperatures Entry';
        f.msg_line2 :=
            'Enter Temperatures - F1 to SAVE or F2 to
ROLLBACK';
        f.state := state;
        f.city := city;
        f.year := year;

        loop := 1;
        loop2 := 0;

        while loop = 1 do
        begin

            display f
                displaying *
                cursor on m1
                accepting m1, m2, m3, m4, m5, m6, m7, m8, m9,
                 m10, m11, m12;


            case f.TERMINATOR of
                PYXIS_$KEY_PF1:
```

```
            loop := 0;
            PYXIS_$KEY_PF2:
            begin
            loop2 := 1;
            loop := 0;
            end;
            OTHERWISE
            f.msg_line2 :=
             'INVALID KEY - F1 to SAVE or F2 to ROLLBACK';
        end;
    end;

    (* if PF2 isn't pressed then update values in *)
    (* database                                    *)

    if loop2 <> 1 then
    begin
        (* update database using values in form *)

        for (TRANSACTION_HANDLE trans) c in cities
             with c.city = city and c.state = state
            modify c using
            c.temperature_array[year,1] := f.m1;
            c.temperature_array[year,2] := f.m2;
            c.temperature_array[year,3] := f.m3;
            c.temperature_array[year,4] := f.m4;
            c.temperature_array[year,5] := f.m5;
            c.temperature_array[year,6] := f.m6;
            c.temperature_array[year,7] := f.m7;
            c.temperature_array[year,8] := f.m8;
            c.temperature_array[year,9] := f.m9;
            c.temperature_array[year,10] := f.m10;
            c.temperature_array[year,11] := f.m11;
            c.temperature_array[year,12] := f.m12;
            end_modify;
        end_for;
    end;

end_form;

get_year (TEMP);
end;
```

```
    COMMIT trans;

end;
(*************************************************************)
(*                                                           *)
(* function to total array for one year (index) and          *)
(* return that to the caller.                                *)
(*                                                           *)
(*************************************************************)
function tot_array (var array_1 : univ mult_array; index :
integer) : double;
var
   i : integer;
   accum : double;
begin
   accum := 0.0;

   for i := 1 to 12 do
       accum := accum + array_1[index,i];

   tot_array := accum;
end;
(*************************************************************)
(*                                                           *)
(* function to total array for one year (index) and          *)
(* return the average to the caller.                         *)
(*                                                           *)
(*************************************************************)
function avg_array (var array_1 : univ mult_array; index :
integer) : double;
var
   i : integer;
   accum : double;
begin

   accum := 0.0;

   for i := 1 to 12 do
       accum := accum + array_1[index,i];

   avg_array := accum / 12.0;

end;
```

```
(*************************************************************)
(*                                                           *)
(* function to total array for one month (index) and         *)
(* return the average to the caller                          *)
(*                                                           *)
(*************************************************************)
function avg_month (var array_1 : univ mult_array; index :
integer) : double;
var
   i : integer;
   accum : double;
begin
   accum := 0.0;

   for i := 1973 to 1975 do
       accum := accum + array_1[i,index];

   avg_month := accum / 3.0;

end;
(*************************************************************)
(*                                                           *)
(* print a report for a city and state combination that       *)
(* includes all the rain fall and temperature information    *)
(* for 1973 through 1975.                                    *)
(*                                                           *)
(*************************************************************)
procedure reports;
const
   under_string  = '                -----  -----  -----  -----
-----  -----  -----  -----  -----  -----  -----  -----   -----
-  ------';
   under_string2 = '                -----  -----  -----  -----
-----  -----  -----  -----  -----  -----  -----  -----';
var
   year,
   i,
   j,
   loop,
   loop2 : integer;
   status : integer;
   result : double;
begin
```

```
    loop := 1;
    loop2 := 0;

    trans := nil;

    START_TRANSACTION trans;

    (* go and get a city & state to operate on *)

    status := get_city_state;

    if status = -1 then
    begin
        return;
    end;

    for i := sizeof (city) downto 1 do
        if city[i] <> ' ' then exit;

    for (TRANSACTION_HANDLE trans) first 1 c in cities with c.city
= city and c.state = state

        writeln;
        writeln;
        write ('                                                ');
        writeln('Temperature and Rain Fall Report');
        writeln;
        writeln;
        write ('                                                    ');
        writeln(city : i, ', ', state);
        writeln;
        writeln;

        write ('        ');
        write ('        ');
        writeln ('  Jan    Feb    Mar    Apr    May    Jun    Jul
Aug     Sep    Oct    Nov    Dec    Total    Avg.');
        write ('        ');
        write ('        ');
        writeln ('-----------------------------------------------
');
        writeln('Rain Fall');
        for j := 1973 to 1975 do
```

```
begin
    write ('         ');
    write (j:4);
    write ('     ');
    write (c.rain_array[j,1] : 5 : 2, '  ');
    write (c.rain_array[j,2] : 5 : 2, '  ');
    write (c.rain_array[j,3] : 5 : 2, '  ');
    write (c.rain_array[j,4] : 5 : 2, '  ');
    write (c.rain_array[j,5] : 5 : 2, '  ');
    write (c.rain_array[j,6] : 5 : 2, '  ');
    write (c.rain_array[j,7] : 5 : 2, '  ');
    write (c.rain_array[j,8] : 5 : 2, '  ');
    write (c.rain_array[j,9] : 5 : 2, '  ');
    write (c.rain_array[j,10] : 5 : 2, '  ');
    write (c.rain_array[j,11] : 5 : 2, '  ');
    write (c.rain_array[j,12] : 5 : 2, '  ');

    result := tot_array(c.rain_array, j);
    write (' ', result : 6 : 2);
    result := avg_array(c.rain_array, j);
    write ('  ', result : 6 : 2);
    writeln;
end;
writeln (under_string);

for j := 1 to 12 do
begin
    rain_array[j] := avg_month(c.rain_array, j);
end;

write ('         ');
write ('Avg      ');
write (rain_array[1] : 5 : 2, '  ');
write (rain_array[2] : 5 : 2, '  ');
write (rain_array[3] : 5 : 2, '  ');
write (rain_array[4] : 5 : 2, '  ');
write (rain_array[5] : 5 : 2, '  ');
write (rain_array[6] : 5 : 2, '  ');
write (rain_array[7] : 5 : 2, '  ');
write (rain_array[8] : 5 : 2, '  ');
write (rain_array[9] : 5 : 2, '  ');
write (rain_array[10] : 5 : 2, '  ');
write (rain_array[11] : 5 : 2, '  ');
```

```
        write (rain_array[12] : 5 : 2, '  ');
        writeln;

        writeln;

        writeln('Temperatures');
        for j := 1973 to 1975 do
        begin
            write ('        ');
            write (j:4);
            write ('     ');
            write (c.temperature_array[j,1] : 5 : 2, '  ');
            write (c.temperature_array[j,2] : 5 : 2, '  ');
            write (c.temperature_array[j,3] : 5 : 2, '  ');
            write (c.temperature_array[j,4] : 5 : 2, '  ');
            write (c.temperature_array[j,5] : 5 : 2, '  ');
            write (c.temperature_array[j,6] : 5 : 2, '  ');
            write (c.temperature_array[j,7] : 5 : 2, '  ');
            write (c.temperature_array[j,8] : 5 : 2, '  ');
            write (c.temperature_array[j,9] : 5 : 2, '  ');
            write (c.temperature_array[j,10] : 5 : 2, '  ');
            write (c.temperature_array[j,11] : 5 : 2, '  ');
            write (c.temperature_array[j,12] : 5 : 2, '  ');

            result := tot_array(c.temperature_array, j);
            write (' ', result : 6 : 2);
            result := avg_array(c.temperature_array, j);
            write ('  ', result : 6 : 2);
            writeln;
        end;
        writeln (under_string);

        for j := 1 to 12 do
        begin
            temperature_array[j] :=
avg_month(c.temperature_array, j);
        end;

        write ('        ');
        write ('Avg      ');
        write (temperature_array[1] : 5 : 2, '  ');
        write (temperature_array[2] : 5 : 2, '  ');
        write (temperature_array[3] : 5 : 2, '  ');
```

```
        write (temperature_array[4] : 5 : 2, '  ');
        write (temperature_array[5] : 5 : 2, '  ');
        write (temperature_array[6] : 5 : 2, '  ');
        write (temperature_array[7] : 5 : 2, '  ');
        write (temperature_array[8] : 5 : 2, '  ');
        write (temperature_array[9] : 5 : 2, '  ');
        write (temperature_array[10] : 5 : 2, '  ');
        write (temperature_array[11] : 5 : 2, '  ');
        write (temperature_array[12] : 5 : 2, '  ');
        writeln;

        writeln;
        writeln;

    end_for;

    COMMIT trans;

end;
(*************************************************************)
(*                                                           *)
(* M A I N    R O U T I N E                                  *)
(*                                                           *)
(*************************************************************)
begin
    cont := 1;

    ready;

    start_transaction;

          (* loop at main menu until EXIT is choosen *)

    while (cont = 1) do
    begin

        case_menu "Array Demo - Pick an Operation"

            menu_entree         "Store / Modify Temperatures" :
                mod_temp();

            menu_entree         "Store / Modify Rain Fall" :
                mod_rain();
```

```
            menu_entree         "Report" :
                reports();

            menu_entree         "Exit"         :
                cont := 0;
        end_menu;

    end;

    commit;

    finish;
end.
```

PL/1 Example

Tourist Information

```
update_guide:procedure options (main);

 /*
 * It's time to update and print the tourist information
 * we have for New England and the Pacific Northwest.  This
 * program first prints and optionally modifies the tourist
 * information for each state. It then creates a database
 * that  is a guide to Northern US seacoasts.
 */

database atlas = filename 'atlas.gdb';
database guide = filename 'coastal_guide.gdb';

%replace     ss$normal by 1;
declare      state_count fixed binary static readonly initial
('08'B4);
declare      state_list (8) character (12) static readonly
              initial
              ('ME', 'NH', 'MA', 'RI', 'CT', 'OR', 'WA', 'AK');

declare lib$get_input ENTRY
          (character (*), character (*), fixed binary (15))
          options (variable) returns (fixed binary (31));

clean_up :procedure;

/*************************************
 *
 *   c l e a n _ u p
 *
 *************************************
 *
 * Functional description
 *
 * At the users' choice, edit the tourism blob
 * for selected states, load a new one from a named
 * file, or accept text from standard input.
 *************************************/

declare filename character (40),
  choice  character (1),
```

```
   i       fixed binary,
   update_tr pointer,
   str_length fixed binary (15),
   stat    fixed binary (31);

based on atlas.states.state state;

/* start a named transaction for update,
   reserving the necessary relations. */

update_tr = null();
start_transaction update_tr concurrency read_write
   reserving atlas.tourism for write,
   atlas.states for read;

do i = 1 to state_count ;
   state = state_list (i);
   for (transaction_handle update_tr) s in states
       cross t in atlas.tourism over state with s.state = state
       put list ('Here''s the information on ', s.state_name)
skip;
       call blob_$display (t.guidebook, atlas, update_tr,
           'guide_book', length ('guidebook'));
       stat = lib$get_input (choice,
                  'Is this information up to date? (Y/N) ');
       if ((choice ~= 'y') & (choice ~= 'Y')) then
       modify t using
           stat = lib$get_input (choice,
   'Enter F to update from a file, E to edit the description ');
           if ((choice = 'F') | (choice = 'f')) then
           do;
               stat = lib$get_input (filename,
                   'Enter full name of input file ',
                   str_length);
               call blob_$load (t.guidebook, atlas,
                   update_tr, filename, str_length);
           end;
           else if ((choice = 'E') | (choice  = 'e')) then
               call blob_$edit (t.guidebook, atlas,
                   update_tr, 'guidebook',
                   length ('guidebook'));
           else
           do;
               PUT LIST
           ('Enter new information from standard input.  ');
```

```
          PUT LIST ('Terminate with <EOF>.') SKIP;
             create_blob new in t.guidebook;
             stat = ss$normal;
             do while (stat = ss$normal);
                 stat = lib$get_input (new.segment,
                 ' ', new.length);
                 put_segment new;
             end;
             close_blob new;
          end;
       end_modify;
   end_for;
end;
commit update_tr;
end clean_up;

transfer_data: procedure ;

/************************************
 *
 *   t r a n s f e r _ d a t a
 *
 ************************************
 *
 * Functional description
 *
 * Move the tourism information for selected
 * states to the guide_book database.
 ************************************/

declaretrans_tr pointer,
   i      binary;
based on atlas.states.statestate;

trans_tr = null ();

start_transaction trans_tr concurrency read_write
   reserving atlas.tourism for read,
   guide.tourism for write;

do i = 1 to state_count;
   for (transaction_handle trans_tr) s in atlas.states
       cross t in atlas.tourism
       over state with s.state = state_list(i)
       do;
```

```
        store (transaction_handle trans_tr) new in guide.tourism
         using
            new.state_name = s.state_name;
            new.state = s.state;
            new.city = t.city;
             new.zip = t.zip;

            /* One way to copy a blob */
            create_blob n_addr in new.office;
            for o_addr in t.office
                n_addr.segment = o_addr.segment;
                n_addr.length = o_addr.length;
                put_segment n_addr;
            end_for;
            close_blob n_addr;

            /* another way to copy a blob */
            call blob_$dump (t.guidebook, atlas, trans_tr,
                    'temp.tmp', length ('temp.tmp'));
            call blob_$load (new.guidebook, guide, trans_tr,
                    'temp.tmp', length ('temp.tmp'));
        end_store;
        end;
    end_for;
    end;

begin;
/*************************************
 *
 *   m a i n
 *
 *************************************
 *
 * Functional description
 *
 * The main routine establishes a list of
 * states from the main atlas database. Its
 * subroutines allow the user to update guidebook
 * descriptions of those states, and store them
 * in a separate database.
 *************************************/

ready atlas;
call clean_up;
```

```
ready  'north_coast_guide.gdb' as guide;
call transfer_data;

finish atlas;
finish guide;
end;
end update_guide;
```

Chapter 3
SQL Examples

This chapter lists an SQL program called Census Report in the following supported host languages:

- Ada
- BASIC
- C
- COBOL
- FORTRAN
- Pascal
- PL/1

Overview

The Census Report program reports on information in the POPULATIONS table. Because data for some of the census years is missing, the program stores values for the missing censuses by using the existing census data in arithmetic calculations. After

reporting on the real and extrapolated population data, the program rolls back the transaction, and thus deletes the calculated data.

The programs in this chapter are available online in the InterBase examples directory, under the name *sql.extension*, where extension is the language extension listed in Chatper 1, *Introduction*.

ADA Example

This example applies to all supported implementations of Ada.

```
WITH basic_io, interbase;

PROCEDURE sql IS

---
--- CENSUS REPORT
---
--- Include the SQL Communications Area.

EXEC SQL
  INCLUDE SQLCA

---
--- Set up to catch SQL errors. Note: this will require that
--- each routine containing executeable SQL commands
--- has an error handling section labeled '999'.
---
EXEC SQL
  WHENEVER SQLERROR GO TO Error_processing;

PROCEDURE print_error IS

---
--- Print out error message.
---
BEGIN --- print_error
  basic_io.put ("Data base error, SQLCODE = ");
  basic_io.put (SQLCODE);
END print_error;

FUNCTION setup RETURN INTEGER IS

---
--- Put in estimated data for missing 1960 & 1970 census data.
--- Return 1 if successful, 0 otherwise.
---
BEGIN --- setup

--- Replace missing 1960 census data with estimates.
  EXEC SQL
      UPDATE POPULATIONS
      SET CENSUS_1960 = (0.3 * (CENSUS_1980 - CENSUS_1950))
```

```
        + CENSUS_1950
       WHERE CENSUS_1960 IS NULL;

--- Replace missing 1970 census data with estimates.
   EXEC SQL
       UPDATE POPULATIONS
       SET CENSUS_1970 = (0.65 * (CENSUS_1980 - CENSUS_1950))
        + CENSUS_1950
       WHERE CENSUS_1970 IS NULL;

   RETURN 1;

<<Error_processing>>
   IF SQLCODE /= 0 THEN
       print_error;
       RETURN 0;
   END IF;
END setup;

PROCEDURE census_report IS

--- Declare variables to receive data retrieved by SQL
--- commands.
pop_50, pop_60, pop_70, pop_80: INTEGER;
stname: BASED_ON STATES.STATE_NAME;

BEGIN --- census report

--- Declare cursor for use in retrieving state/census data.
     EXEC SQL
          DECLARE C CURSOR FOR
          SELECT STATE_NAME, CENSUS_1950, CENSUS_1960,
              CENSUS_1970, CENSUS_1980
          FROM STATES S, POPULATIONS P
          WHERE S.STATE = P.STATE
          ORDER S.STATE_NAME;

--- Open cursor for use in retrieving state/census data.
   EXEC SQL
       OPEN C;

--- Print report header
   basic_io.put ("  STATE                    POPULATION");
   basic_io.new_line;
   basic_io.put ("                                   ");
```

```
   basic_io.put ("1950       1960       1970       1980");
   basic_io.new_line;

--- Retrieve first set of state/census data
   EXEC SQL
       FETCH C INTO :stname, :pop_50, :pop_60, :pop_70, :pop_80;

--- Until end of data stream is reached, print out current set
--- of data and then fetch next set.

   WHILE SQLCODE = 0 LOOP
       basic_io.put (stname);
       basic_io.put (pop_50);
       basic_io.put (pop_60);
       basic_io.put (pop_70);
       basic_io.put (pop_80);
       basic_io.new_line;

--- Retrieve next set of state/census data.
       EXEC SQL
           FETCH C INTO :stname, :pop_50, :pop_60,
                :pop_70, :pop_80;
   END LOOP;

--- Close cursor
   EXEC SQL
       CLOSE C;

<<Error_processing>>
   IF SQLCODE /= 0 THEN
       print_error;
   END IF;
END census_report;

BEGIN --- sql

--- Set up estimated data; if successful then do report.
   IF setup = 1 THEN
       census_report;
   END IF;

--- Undo estimated census data.
   EXEC SQL
       ROLLBACK;
```

```
--- Output error information.
<<Error_processing>>
   IF SQLCODE /= 0 THEN
       print_error;
   END IF;
END sql;
```

BASIC Example

```
10  %TITLE "TEST"

    ! CENSUS REPORT

    DECLARE WORD FUNCTION SETUPDATA

    ! Include the SQL Communications Area.
    EXEC SQL
        INCLUDE SQLCA

    ! Set up to catch SQL errors.
    EXEC SQL
        WHENEVER SQLERROR GO TO 99

    ! Set up estimated data. If successful,generate report.
    CALL SETUPDATA
    IF SQLCODE = 0 THEN CALL CENSUS_REPORT END IF

    ! Undo estimated census data.
    EXEC SQL
        ROLLBACK

    ! Output error information, if any.
99  IF SQLCODE <> 0 THEN CALL PRINT_ERROR END IF
    END

    ! Print out error message.
200 SUB PRINT_ERROR

    ! Include the SQL Communications Area.
    EXEC SQL
        INCLUDE SQLCA

    PRINT 'Data base error, SQLCODE = ', SQLCODE
    RETURN
    END SUB

    ! Put in estimated data for missing 1960 & 1970 census
      data.Return 1 if successful, 0 otherwise.
300 SUB SETUPDATA

    ! Include the SQL Communications Area.
    EXEC SQL
        INCLUDE SQLCA
```

```
    ! Set up to catch SQL errors.
    EXEC SQL
        WHENEVER SQLERROR GO TO 399

    ! Replace missing 1960 census data with estimates.
    EXEC SQL &
        UPDATE POPULATIONS &
        SET CENSUS_1960 = &
            (0.3 * (CENSUS_1980 - CENSUS_1950)) + CENSUS_1950 &
            WHERE CENSUS_1960 IS NULL

    ! Replace missing 1970 census data with estimates.
    EXEC SQL &
        UPDATE POPULATIONS &
        SET CENSUS_1970 = &
            (0.65 * (CENSUS_1980 - CENSUS_1950)) + CENSUS_1950 &
            WHERE CENSUS_1970 IS NULL

    EXIT SUB

    ! Error return
399 CALL PRINT_ERROR
    EXIT SUB
    END SUB

400 SUB CENSUS_REPORT

    ! Declare variables to receive data retrieved by SQL
      commands.
    DECLARE LONG POP_50, POP_60, POP_70, POP_80
    DECLARE STRING STNAME

    ! Include the SQL Communications Area.
    EXEC SQL
        INCLUDE SQLCA

    ! Set up to catch SQL errors.
    EXEC SQL
        WHENEVER SQLERROR GO TO 499

    ! Declare cursor for use in retrieving state/census data.
         EXEC SQL &
            DECLARE C CURSOR FOR &
            SELECT STATE_NAME, CENSUS_1950, CENSUS_1960, &
                CENSUS_1970, CENSUS_1980 &
            FROM STATES S, POPULATIONS P &
```

```
        WHERE S.STATE = P.STATE &
        ORDER S.STATE_NAME

  ! Open cursor for use in retrieving state/census data.
  EXEC SQL &
      OPEN C

  ! Print report header.
  PRINT 'STATE                    ', &
      '               POPULATION'
  PRINT '                                  1950', &
      '   1960          1970          1980'

  ! Retrieve first set of state/census data.
  EXEC SQL &
      FETCH C INTO :STNAME, :POP_50, :POP_60, :POP_70, :POP_80

  ! Until end of data stream is reached, print out current
    set of data and then fetch next set.
  WHILE SQLCODE = 0
              PRINT STNAME, POP_50, POP_60, POP_70, POP_80

      ! Retrieve next set of state/census data.
      EXEC SQL &
          FETCH C INTO :STNAME, :POP_50, :POP_60, &
               :POP_70, :POP_80
      NEXT

  ! Close cursor.
  EXEC SQL &
      CLOSE C

  EXIT SUB

  ! Error return
499 CALL PRINT_ERROR
  EXIT SUB
  END SUB
```

C Example

Be sure to use the **-e** (either case) switch on **gpre** when precompiling this program.

```
/* CENSUS REPORT */

/* Include the SQL Communications Area. */
EXEC SQL
  INCLUDE SQLCA

/* Set up to catch SQL errors. Note: this will require that each
routine containing executeable SQL commands has
an error handling section labeled 'err'.*/
EXEC SQL
  WHENEVER SQLERROR GO TO err;

main()
{
  /* Set up estimated data; if successful generate report. */
  if (setup()) census_report();

  /* Undo estimated census data. */
  EXEC SQL
      ROLLBACK;

  return;

  /* Output error information. */
err:  print_error();
}

void print_error()

/* Print out error message */
{
printf ("Data base error, SQLCODE = %d\n", SQLCODE);
}

int setup()
/* Put in estimated data for missing 1960 & 1970 census data.
/* Return 1 if successful, 0 otherwise.  */
{
  /* Replace missing 1960 census data with estimates. */
  EXEC SQL
      UPDATE POPULATIONS
      SET CENSUS_1960 =
```

```
        (0.3 * (CENSUS_1980 - CENSUS_1950)) + CENSUS_1950
        WHERE CENSUS_1960 IS NULL;

  /* Replace missing 1970 census data with estimates. */
  EXEC SQL
      UPDATE POPULATIONS
      SET CENSUS_1970 =
          (0.65 * (CENSUS_1980 - CENSUS_1950)) + CENSUS_1950
          WHERE CENSUS_1970 IS NULL;

  return (1);

  /* Error return */
err:print_error();
  return (0);
}

void census_report()
{
  /* Declare variables to receive data retrieved by SQL
  commands. */
  int pop_50, pop_60, pop_70, pop_80;
  char stname[26];

  /* Declare cursor for use in retrieving state/census data. */
  EXEC SQL
      DECLARE C CURSOR FOR
      SELECT STATE_NAME, CENSUS_1950, CENSUS_1960,
          CENSUS_1970, CENSUS_1980
      FROM STATES S, POPULATIONS P
      WHERE S.STATE = P.STATE
      ORDER S.STATE_NAME;

  /* Open cursor for use in retrieving state/census data. */
  EXEC SQL
      OPEN C;

  /* Print report header */
  printf ("     STATE                    ");
  printf ("                  POPULATION\n");
  printf ("                                  1950        1960");
  printf ("        1970        1980\n\n");

  /* Retrieve first set of state/census data. */
  EXEC SQL
      FETCH C INTO :stname, :pop_50, :pop_60, :pop_70, :pop_80;
```

```
    /* Until end of data stream is reached, print out current set
       of data and then fetch next set. */
    while (SQLCODE == 0)
        {
                  printf ("%-25s   %9d   %9d   %9d   %9d\n",
            stname, pop_50,
            pop_60, pop_70, pop_80);

        /* Retrieve next set of state/census data. */
        EXEC SQL
            FETCH C INTO :stname, :pop_50, :pop_60,
                 :pop_70, :pop_80;
        }

    /* Close cursor. */
    EXEC SQL
        CLOSE C;

    return;

    /* Error return */
err:  print_error();
    return;
}
```

COBOL Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. C_REPORT.

*
* CENSUS_REPORT
*
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
* Include the SQL Communications Area
    EXEC SQL
         INCLUDE SQLCA
    END-EXEC

* Declare variables to receive data retrieved by SQL
commands.
01  STNAME PIC X(25).
01  POP_50 PIC S9(9) USAGE COMP.
01  POP_60 PIC S9(9) USAGE COMP.
01  POP_70 PIC S9(9) USAGE COMP.
01  POP_80 PIC S9(9) USAGE COMP.
01  D_POP_50 PIC Z(9).
01  D_POP_60 PIC Z(9).
01  D_POP_70 PIC Z(9).
01  D_POP_80 PIC Z(9).
01  D_SQLCODEPIC Z(4).

PROCEDURE DIVISION.
MAIN_ROUTINE.

* Set up to catch SQL errors. Note: this will require that each
* routine containing executeable SQL commands has an error
* handling section labeled 'ERR'.*
  EXEC SQL
      WHENEVER SQLERROR GO TO ERR
  END-EXEC

  PERFORM SETUP.
  IF SQLCODE = 0 THEN PERFORM CENSUS_REPORT.
  EXEC SQL
      ROLLBACK
  END-EXEC
  STOP RUN.
```

```
* Print out error message.
PRINT_ERROR.

  MOVE SQLCODE TO D_SQLCODE.
  DISPLAY "Data base error, SQLCODE = ", D_SQLCODE.

SETUP.

* Put in estimated data for missing 1960 & 1970 census data.
* Return 1 if successful, 0 otherwise.

* Replace missing 1960 census data with estimates.
  EXEC SQL
      UPDATE POPULATIONS
      SET CENSUS_1960 =
          (0.3 * (CENSUS_1980 - CENSUS_1950)) + CENSUS_1950
          WHERE CENSUS_1960 IS NULL
  END-EXEC

* Replace missing 1970 census data with estimates.
  EXEC SQL
      UPDATE POPULATIONS
      SET CENSUS_1970 = 0
          (.65 * (CENSUS_1980 - CENSUS_1950)) + CENSUS_1950
          WHERE CENSUS_1970 IS NULL
  END-EXEC.

CENSUS_REPORT.

* Declare cursor for use in retrieving state/census data.
        EXEC SQL
      DECLARE C CURSOR FOR
      SELECT STATE_NAME, CENSUS_1950, CENSUS_1960,
          CENSUS_1970, CENSUS_1980
      FROM STATES S, POPULATIONS P
      WHERE S.STATE = P.STATE
      ORDER S.STATE_NAME
  END-EXEC

* Open cursor for use in retrieving state/census data.
  EXEC SQL
      OPEN C
  END-EXEC

* Print report header.
  DISPLAY "     STATE                ",
```

```
      "                    POPULATION".
   DISPLAY "                                  1950",
         "        1960        1970        1980".
   DISPLAY " ".

*  Retrieve first set of state/census data.
   EXEC SQL
       FETCH C INTO :STNAME, :POP_50, :POP_60, :POP_70, :POP_80
   END-EXEC

*  Until end of data stream is reached, print out current set
*  of data and then fetch next set.
   PERFORM CENSUS_LINE UNTIL SQLCODE NOT = 0.

*  Close cursor.
   EXEC SQL
       CLOSE C
   END-EXEC.

CENSUS_LINE.
   MOVE POP_50 TO D_POP_50.
   MOVE POP_60 TO D_POP_60.
   MOVE POP_70 TO D_POP_70.
   MOVE POP_80 TO D_POP_80.
   DISPLAY STNAME, D_POP_50, "    ", D_POP_60, "    ",
       D_POP_70, "    ", D_POP_80.

*  Retrieve next set of state/census data.
   EXEC SQL
       FETCH C INTO :STNAME, :POP_50, :POP_60,
           :POP_70, :POP_80
   END-EXEC.

*  Error return
ERR.
   EXEC SQL
       WHENEVER SQLERROR CONTINUE
   END-EXEC
   PERFORM PRINT_ERROR.
   EXEC SQL
       ROLLBACK
   END-EXEC
   STOP RUN.
```

FORTRAN Example

This example applies to all supported implementations of FORTRAN.

```
C       CENSUS REPORT

        INTEGER SETUP
C       Include the SQL Communications Area
        EXEC SQL
     *       INCLUDE SQLCA

C       Set up to catch SQL errors. Note: this will require
C          that each routine containing executeable SQL commands
C       has an error handling section labeled '999'.
        EXEC SQL
     *       WHENEVER SQLERROR GO TO 999

C       Set up estimated data; if successful then do report
        IF (SETUP(0) .EQ. 1) CALL CENSUS_REPORT()

C       Undo estimated census data

        EXEC SQL
     *       ROLLBACK

        STOP

C       Output error information
999     CALL PRINT_ERROR()
        STOP
        END

        SUBROUTINE PRINT_ERROR()
C       Include the SQL Communications Area
        EXEC SQL
     *       INCLUDE SQLCA

C       Print out error message
        WRITE (*,*) 'Data base error, SQLCODE = ', SQLCODE
        RETURN
        END

        INTEGER FUNCTION SETUP(DUMMY)
C       Put in estimated data for missing 1960 & 1970 census
```

```
C       data. Return 1 if successful, 0 otherwise.

C       Include the SQL Communications Area
        EXEC SQL
     *       INCLUDE SQLCA

C        Replace missing 1960 census data with estimates

        EXEC SQL
     *       UPDATE POPULATIONS
     *       SET CENSUS_1960 = (0.3 * (CENSUS_1980 -
     *                          CENSUS_1950))
     *                        + CENSUS_1950
     *       WHERE CENSUS_1960 IS NULL

C       Replace missing 1970 census data with estimates

        EXEC SQL
     *       UPDATE POPULATIONS
     *       SET CENSUS_1970 = (0.65 * (CENSUS_1980 -
     *                          CENSUS_1950))
     *                       + CENSUS_1950
     *       WHERE CENSUS_1970 IS NULL

        SETUP = 1
        RETURN

C       Error return
999     CALL PRINT_ERROR()
        SETUP = 0
        RETURN
        END

        SUBROUTINE CENSUS_REPORT()
C       Declare variables to receive data retrieved by SQL
C       commands
        INTEGER POP_50, POP_60, POP_70, POP_80
        CHARACTER*25 STNAME

C       Include the SQL Communications Area
        EXEC SQL
     *       INCLUDE SQLCA
```

```
C       Declare cursor for use in retrieving state/census data
        EXEC SQL
      *       DECLARE C CURSOR FOR
      *       SELECT STATE_NAME, CENSUS_1950, CENSUS_1960,
      *           CENSUS_1970, CENSUS_1980
      *       FROM STATES S, POPULATIONS P
      *       WHERE S.STATE = P.STATE
      *       ORDER S.STATE_NAME

C       Open cursor for use in retrieving state/census data
        EXEC SQL
      *       OPEN C

C       Print report header
        WRITE (*,1001) 'STATE', 'POPULATION'
1001    FORMAT (T5,A,T42,A)
        WRITE (*,1002) '1950', '1960', '1970', '1980'
1002    FORMAT (T29,A,T41,A,T53,A,T65,A)

C       Retrieve first set of state/census data
        EXEC SQL
       *       FETCH C INTO :STNAME, :POP_50, :POP_60, :POP_70,
      *            :POP_80

C       Until end of data stream is reached, print out current
C       set of data and then fetch next set
        DO WHILE (SQLCODE .EQ. 0)
            WRITE (*,1000) STNAME, POP_50, POP_60, POP_70,
       * POP_80
1000        FORMAT (1X,A25,T26,I9,T38,I9,T50,I9,T62,I9)
C           Retrieve next set of state/census data
            EXEC SQL
       *           FETCH C INTO :STNAME, :POP_50, :POP_60,
       *              :POP_70, :POP_80
        END DO

C       Close cursor
        EXEC SQL
       *       CLOSE C

        RETURN

C       Error return
```

```
999     CALL PRINT_ERROR()
        RETURN
        END
```

Pascal Example

This example applies to all supported implementations of Pascal. For formatting purposes, quoted strings in this example sometimes wrap on multiple lines. When you code this program, be sure to put the quoted strings on a single line.

```
program test (OUTPUT);

(* CENSUS REPORT *)

(* Include the SQL Communications Area *)
EXEC SQL
  INCLUDE SQLCA

(* Set up to catch SQL errors. Note: this will require that
   each routine containing executable SQL commands has an
   error handling section labeled '999'. *)

EXEC SQL
  WHENEVER SQLERROR GO TO 999
label 999;

procedure print_error;
(* Print out error message *)
begin
  writeln ('Data base error, SQLCODE = ', SQLCODE);
  gds_$print_status (gds_$status);
end; (* print_error *)

function setup: integer;
(* Put in estimated data for missing 1960 & 1970 census data.
   Return 1 if successful, 0 otherwise.  *)
label 999;
begin
  (* Replace missing 1960 census data with estimates *)

  EXEC SQL
      UPDATE POPULATIONS
      SET CENSUS_1960 = (0.3 * (CENSUS_1980 - CENSUS_1950)) +
      CENSUS_1950
      WHERE CENSUS_1960 IS NULL;

  (* Replace missing 1970 census data with estimates *)
```

```
   EXEC SQL
       UPDATE POPULATIONS
       SET CENSUS_1970 = (0.65 * (CENSUS_1980 - CENSUS_1950)) +
           CENSUS_1950
       WHERE CENSUS_1970 IS NULL;

   setup := 1;

999: if SQLCODE <> 0 then
       begin
       print_error;
       setup := 0;
       end;
end; (* setup *)

procedure census_report;
   (* Declare variables to receive data retrieved by
      SQL commands *)
var
(* for the Apollo version of Pascal, this declaration must
   be integer32 *)
   pop_50, pop_60, pop_70, pop_80: integer;
   stname: packed array [1..25] of char;
label 999;
begin
   (* Declare cursor for use in retrieving state/census data *)
  EXEC SQL
       DECLARE C CURSOR FOR
       SELECT STATE_NAME, CENSUS_1950, CENSUS_1960,
          CENSUS_1970, CENSUS_1980
       FROM STATES S, POPULATIONS P
       WHERE S.STATE = P.STATE
       ORDER S.STATE_NAME;

   (* Open cursor for use in retrieving state/census data *)
   EXEC SQL
       OPEN C;

   (* Print report header *)
         writeln ('    STATE                ',
       '                    POPULATION');
   writeln ('                          ', '1950':10,
       '1960':10,'1970':10,'1980':10);
```

```
    writeln;

    (* Retrieve first set of state/census data *)
    EXEC SQL
        FETCH C INTO :stname, :pop_50, :pop_60, :pop_70, :pop_80;

    (* Until end of data stream is reached, print out current set
       of data and then fetch next set *)
    while SQLCODE = 0 do
        begin
                writeln (stname:25, pop_50:9, ' ', pop_60:9 ,' ',
            pop_70:9, ' ', pop_80:9);

        (* Retrieve next set of state/census data *)
        EXEC SQL
            FETCH C INTO :stname, :pop_50, :pop_60,
                :pop_70, :pop_80;
        end;

    (* Close cursor *)
    EXEC SQL
        CLOSE C;

999:if SQLCODE <> 0 then print_error;
end; (* census_report *)

begin

(* Set up estimated data; if successful then do report *)
   if setup = 1 then census_report;

(* Undo estimated census data *)
EXEC SQL
  ROLLBACK;


{* Output error information *)
999: if SQLCODE <> 0 then print_error;
end.
```

PL/1 Example

```
  /* CENSUS REPORT */

  /* Include the SQL Communications Area. */
  EXEC SQL
      INCLUDE SQLCA

  /* Set up to catch SQL errors. Note: this will require that
      each routine containing executeable SQL commands has
      an error handling section labeled 'ERR'. */
  EXEC SQL
      WHENEVER SQLERROR GO TO ERR;

  /* Print out error message */
  PRINT_ERROR: PROCEDURE;
  PUT LIST ('Data base error, SQLCODE = ', SQLCODE) SKIP;
  END PRINT_ERROR;

SETUP: PROCEDURE RETURNS (FIXED BINARY (15));

  /* Put in estimated data for missing 1960 & 1970 census data.
     Return 1 if successful, 0 otherwise.  */

  /* Replace missing 1960 census data with estimates */
  EXEC SQL
      UPDATE POPULATIONS
      SET CENSUS_1960 =
          (0.3 * (CENSUS_1980 - CENSUS_1950)) + CENSUS_1950
      WHERE CENSUS_1960 IS NULL;

  /* Replace missing 1970 census data with estimates */
  EXEC SQL
      UPDATE POPULATIONS
      SET CENSUS_1970 =
          (0.65 * (CENSUS_1980 - CENSUS_1950)) + CENSUS_1950
      WHERE CENSUS_1970 IS NULL;

  RETURN (1);

  /* Error return */
ERR: CALL PRINT_ERROR;
     RETURN (0);
     END SETUP;

CENSUS_REPORT: PROCEDURE;
```

```
/* Declare variables to receive data retrieved by SQL
    commands. */
DECLARE (POP_50, POP_60, POP_70, POP_80) FIXED BINARY (31);
DECLARE STNAME CHARACTER(25);

/* Declare cursor for use in retrieving state/census data. */
EXEC SQL
    DECLARE C CURSOR FOR
    SELECT STATE_NAME, CENSUS_1950, CENSUS_1960,
        CENSUS_1970, CENSUS_1980
    FROM STATES S, POPULATIONS P
    WHERE S.STATE = P.STATE
    ORDER S.STATE_NAME;

/* Open cursor for use in retrieving state/census data. */
EXEC SQL
    OPEN C;

/* Print report header. */
      PUT LIST ('     STATE                ') SKIP;
PUT LIST ('              POPULATION');
PUT LIST ('                                  1950') SKIP;
      PUT LIST ('   1960        1970        1980');
PUT LIST (' ') SKIP;

/* Retrieve first set of state/census data. */
EXEC SQL
    FETCH C INTO :STNAME, :POP_50, :POP_60, :POP_70, :POP_80;

/* Until end of data stream is reached, print out current
   set of data,then fetch next set. */
DO WHILE (SQLCODE = 0);
              PUT EDIT (STNAME, POP_50,
        POP_60, POP_70, POP_80)
        (A(25), 4 (X(3), F(9))) SKIP;

    /* Retrieve next set of state/census data. */
    EXEC SQL
        FETCH C INTO :STNAME, :POP_50, :POP_60,
            :POP_70, :POP_80;
    END;

/* Close curso.r */
EXEC SQL
    CLOSE C;
```

```
   RETURN;

   /* Error return */
ERR: CALL PRINT_ERROR;
     RETURN;
     END CENSUS_REPORT;

   /* Set up estimated data; if successful, generate report. */
   IF (SETUP() = 1) THEN CALL CENSUS_REPORT();

   /* Undo estimated census data. */
   EXEC SQL
       ROLLBACK;

   RETURN;

   /* Output error information. */
ERR: CALL PRINT_ERROR;
     END;
```

Chapter 4
DSQL Examples

This chapter lists a DSQL program called Executing a Known Statement in the following supported host languages:

- Ada
- BASIC
- C
- COBOL
- FORTRAN
- Pascal
- PL/1

It also lists a DSQL program called Executing an Unknown Statement in C and Pascal.

Overview

An overview of the DSQL example programs is presented below. For step-by-step information on writing a DSQL program, refer to the DSQL part of the *Programmer's Guide*.

Executing a Known Statement

The Executing a Known Statement program shows how to write a simple DSQL program. It uses a string constant as a query in order to highlight the following steps:

- Declaring variables
- Setting up an SQLDA
- Executing a DSQL statement

In a real application, you would develop the query string through some sort of interaction with the user.

This program is available online in the InterBase examples directory under the name dsql.*extension,* where *extension* is the language extension listed in Chapter 1, *Introduction*.

Executing an Unknown Statement

The Executing an Unknown Statement program shows how to write a DSQL program that can execute a statement about which the program has no prior information. This program assumes that the statements do not have input parameters. It also uses the non-SQL **database** declaration, **ready** command, and **start_transaction** command, in order to work with an unknown database.

This program is available online in the InterBase examples directory under the name full_dsql.*extension,* where *extension* is the language extension listed in Chapter 1, *Introduction*.

Ada Example

Executing a Known Statement

This example applies to all supported implementations of Ada.

```
WITH basic_io, interbase;

PROCEDURE dsql IS

EXEC SQL
   INCLUDE SQLCA

-- *  Set up the query statement
query: CONSTANT string(1..61) := "SELECT CITY, STATE, POPULATION
FROM CITIES WHERE STATE = 'NY'";

-- *  Declare variables to receive data. Note: if it is not
-- *  known beforehand the nature of the fields to be
-- *  retrieved, it would be necessary to allocate
-- *  buffer space for the fields at run time, rather than
-- *  using predeclared fields.

TYPE sqlda IS RECORD
   sqldaid: interbase.chars (1..8);
   sqldabc: integer;
   sqln    : short_integer := 3;
   sqld    : short_integer;
   sqlvars: interbase.sqlvar_array (1..3);
END RECORD;
FOR sqlda use record at mod 2;
   sqldaid at 0 range 0..63;
   sqldabc at 8 range 0..31;
   sqln    at 12 range 0..15;
   sqld    at 14 range 0..15;
   sqlvars at 16 range 0..1055;
END RECORD;

sqlda1: sqlda;

city: BASED_ON CITIES.CITY;
state: BASED_ON CITIES.STATE;
state: BASED_ON CITIES.POPULATION;
```

```
flag1, flag2, flag3: interbase.isc_short;

begin

EXEC SQL
  WHENEVER SQLERROR GO TO Error_processing;

-- *  Prepare the query
    EXEC SQL
        PREPARE Q INTO sqlda1 FROM :query;

-- *  Set up SQLDA to point to declared fields, adjusting
-- *  datatypes and setting null terminators on the way
-- *  through.  Change type for city & state fields to be
-- *  TEXT rather than VARYING since C does not know
-- *  about VARYING fields.  The format <datatype> + 1
--*   indicates that the field is of <datatype> and has
-- *  an indicator variable
    sqlda1.sqlvars(1).sqldata := city'address;
    sqlda1.sqlvars(1).sqlind := flag1'address;
    sqlda1.sqlvars(1).sqltype := interbase.SQL_TEXT + 1;
    sqlda1.sqlvars(2).sqldata := state'address;
    sqlda1.sqlvars(2).sqlind := flag2'address;
    sqlda1.sqlvars(2).sqltype := interbase.SQL_TEXT + 1;
    sqlda1.sqlvars(3).sqldata := population'address;
    sqlda1.sqlvars(3).sqlind := flag3'address;

    EXEC SQL
        DECLARE C CURSOR FOR Q;

    EXEC SQL
        OPEN C;

-- *  Fetch and print out records
    EXEC SQL
        FETCH C USING DESCRIPTOR sqlda1;
    WHILE SQLCODE = 0 LOOP
  basic_io.put (city);
  basic_io.put (state);
  basic_io.put (population);
  basic_io.new_line;
      EXEC SQL
       FETCH C USING DESCRIPTOR sqlda1;
```

```
   END LOOP;

     EXEC SQL
   COMMIT RELEASE;

<<Error_processing>>
     IF SQLCODE /= 0 THEN
   basic_io.put ("Data base error, SQLCODE = ");
   basic_io.put (SQLCODE);
   basic_io.new_line;
     END IF;

END dsql;
```

BASIC Example

Executing a Known Statement

```
10  %TITLE "DSQL"
        !
        ! copyright (c) 1989 by Interbase Software Corporation
        !

    EXTERNAL LONG FUNCTION addr
    DECLARE STRING city, state
    DECLARE STRING CONSTANT query = &
    'SELECT CITY, STATE, POPULATION FROM CITIES
         WHERE STATE = "NY"'
    DECLARE LONG population
    DECLARE WORD flag1, flag2, flag3
    RECORD SQLDA
        STRING SQLDAID = 8
        LONG SQLDABC
        WORD SQLN
        WORD SQLD
        GROUP SQLVAR (2)
            WORD SQLTYPE
            WORD SQLLEN
            LONG SQLDATA
            LONG SQLIND
            WORD SQLNAME_LENGTH
            STRING SQLNAME = 30
              END GROUP SQLVAR
    END RECORD SQLDA
    DECLARE SQLDA SQLDA1

    EXEC SQL INCLUDE SQLCA

    EXEC SQL WHENEVER SQLERROR GO TO 999

    city = SPACE$(25)
          state = SPACE$(2)
          SQLDA1::SQLN = 3
    EXEC SQL PREPARE Q INTO sqlda1 FROM :query
```

```
  ! Establish connection between the point fields in the SQLDA
  ! and the associated program variables.On the way
  ! through, force text fields with SQL_TEXT, since
  ! SQL_VARYING won't work in BASIC

  ! Note that <datatype> + 1 indicates a variable of
  ! <datatype> with an associated indicator field.

       SQLDA1::SQLVAR(0)::SQLDATA = addr (city BY REF)
       SQLDA1::SQLVAR(0)::SQLIND = addr (flag1 BY REF)
       SQLDA1::SQLVAR(0)::SQLTYPE = SQL_TEXT + 1

       SQLDA1::SQLVAR(1)::SQLDATA = addr (state BY REF)
       SQLDA1::SQLVAR(1)::SQLIND = addr (flag2 BY REF)
       SQLDA1::SQLVAR(1)::SQLTYPE = SQL_TEXT + 1

       SQLDA1::SQLVAR(2)::SQLDATA = addr (population BY REF)
       SQLDA1::SQLVAR(2)::SQLIND = addr (flag3 BY REF)

  EXEC SQL DECLARE C CURSOR FOR Q

  EXEC SQL OPEN C

100
  EXEC SQL FETCH C USING DESCRIPTOR sqlda1
  IF SQLCODE <> 0% THEN
          GO TO 200
       END IF
  PRINT city, state, population
  GO TO 100
200
  EXEC SQL COMMIT RELEASE

999     IF SQLCODE <> 0% THEN
          PRINT 'Data base error, SQLCODE = '; SQLCODE
          CALL gds_$print_status (gds_$status() BY REF)
       END IF
  end
```

C Examples

Executing a Known Statement

Be sure to use the **-e** (either case) switch on **gpre** when precompiling this program.

```
    EXEC SQL
         INCLUDE SQLCA

/*  Set up the query statement */
    char *query = "SELECT CITY, STATE, POPULATION
          FROM CITIES WHERE STATE = 'NY'";

main ()
{
/*  Declare variables to receive data. Note: if it is not
 *  known beforehand the nature of the fields to be
 *  retrieved, it would be necessary to allocate
 *  buffer space for the fields at run time, rather than using
 *  predeclared fields. */
    char city[26], state[5];
    long population;
    short flag1, flag2, flag3;

/*  Declare and allocate the SQLDA */
    SQLDA *sqlda;
    sqlda = (SQLDA*) malloc (SQLDA_LENGTH (3));
    sqlda->sqln = 3;

    EXEC SQL
        WHENEVER SQLERROR GO TO ERR;

/*  Prepare the query */
    EXEC SQL
        PREPARE Q INTO sqlda FROM :query;

/*  Set up SQLDA to point to declared fields, adjusting
 *  datatypes and setting null terminators on the way
 *  through. Change type for city & state fields to be
 *  TEXT rather than VARYING since C does not know
 *  about VARYING fields.  The format <datatype> + 1 indicates
 *  that the field is of <datatype> and has an
```

```
 *  indicator variable */
    sqlda->sqlvar[0].sqldata = city;
    city[25] = 0;
    sqlda->sqlvar[0].sqlind = &flag1;
    sqlda->sqlvar[0].sqltype = SQL_TEXT + 1;

    sqlda->sqlvar[1].sqldata = state;
    state[5] = 0;
    sqlda->sqlvar[1].sqlind = &flag2;
    sqlda->sqlvar[1].sqltype = SQL_TEXT + 1;

    sqlda->sqlvar[2].sqldata = (char*) &population;
    sqlda->sqlvar[2].sqlind = &flag3;

    EXEC SQL
        DECLARE C CURSOR FOR Q;

    EXEC SQL
        OPEN C;

/*  Fetch and print out records */
    EXEC SQL
        FETCH C USING DESCRIPTOR sqlda;
    while (SQLCODE == 0)
        {
        printf ("%s %s %d\n", city, state, population);
        EXEC SQL
            FETCH C USING DESCRIPTOR sqlda;
        }

    EXEC SQL
  COMMIT RELEASE;

    exit();

ERR: printf ("Data base error, SQLCODE = \n", SQLCODE);
    gds_$print_status (gds_$status);

}
```

Executing an Unknown Statement

Be sure to use the **-e** (either case) switch on **gpre** when precompiling this program.

```
#include <stdio.h>
#ifdef VMS
#include time.h
#else
#include <sys/time.h>
#endif

extern char*malloc();

DATABASE DB = COMPILETIME "atlas.gdb"
            RUNTIME db_name;

chardb_name[128];
/*************************************
 *
 * Macro for aligning buffer space to boundaries
 *   needed by a datatype on a given machine
 *
 *************************************/
#ifdef VMS
#define ALIGN(n,b)(n)
#endif

#ifdef sun
#ifdef sparc
#define ALIGN(n,b)((n + b - 1) & ~(b - 1))
#endif
#endif

#ifdef hpux
#define ALIGN(n,b)((n + b - 1) & ~(b - 1))
#endif

#ifdef ultrix
#define ALIGN(n,b)((n + b - 1) & ~(b - 1))
#endif

#ifdef AIX
#define ALIGN(n,b)((n + b - 1) & ~(b - 1))
```

```
#endif

#ifndef ALIGN
#define ALIGN(n,b)((n+1) & ~1)
#endif

#define UPPER(c) ((c >= 'a' && c<= 'z') ? c + 'A' - 'a': c )

static char*alpha_months [] =
          {
          "January",
          "February",
          "March",
          "April",
          "May",
          "June",
          "July",
          "August",
          "September",
          "October",
          "November",
          "December"
          };

typedef struct vary {
    shortvary_length;
    charvary_string [1];
} VARY;



main (argc, argv)
    intargc;
    char*argv[];
{
/************************************
 *
 *m a i n
 *
 ************************************
 *
 * Functional description
 *Process sql statements until utterly bored.
```

```
 *
 ************************************/
char  statement [512];
SQLDA  *sqlda;

/* Get database name */

if (argc < 2)
    {
    fprintf (stderr, "No database specified on command
          line\n");
    exit (1);
    }
++argv;
strcpy (db_name, *argv);

/* Open database and start tansaction */

READY;
START_TRANSACTION;

/* Set up SQLDA for SELECTs, allow up to 20 fields to be
  selected */

sqlda = (SQLDA*) malloc (SQLDA_LENGTH (20));
sqlda->sqln = 20;

/* Prompt for SQL statements and process them */

while (1)
    {
    /* get command */
    get_statement (statement);

    /* Check for command to leave */
    if ((UPPER (statement[0]) == 'E') &&
        (UPPER (statement[1]) == 'X') &&
        (UPPER (statement[2]) == 'I') &&
        (UPPER (statement[3]) == 'T'))
         {
   COMMIT;
   FINISH;
   return;
```

```
        }
    if ((UPPER (statement[0]) == 'Q') &&
        (UPPER (statement[1]) == 'U') &&
        (UPPER (statement[2]) == 'I') &&
        (UPPER (statement[3]) == 'T'))
        {
   ROLLBACK;
   FINISH;
   return;
        }

    process_statement (statement, sqlda);
    }
}


static get_statement (statement)
    char *statement;
{
/*************************************
 *
 *g e t _ s t a t e m e n t
 *
 *************************************
 *
 * Functional description
 *Get an SQL statment, or QUIT/EXIT command to process
 *
 *************************************/
char *p;
short  c, blank, done;

blank = 1;
while (blank)
    {
    printf ("SQL> ");
    blank = 1;
    p = statement;
    done = 0;
    while (!done)
        switch (c = getchar ())
            {
            case EOF:
```

```
                strcpy (statement, "EXIT");
                return;

            case '\n':
                if (p == statement || p [-1] != '-')
                    done = 1;
                else
                    {
                    p [-1] = ' ';
                    printf ("CON> ");
                    }
                break;

            case '\t':
            case ' ':
                *p++ = ' ';
                break;

            default:
                *p++ = c;
                blank = 0;
                break;
            }
        *p = 0;
        }
    }


    static print_item (s, var)
        char    **s;
        SQLVAR  *var;
    {
    /************************************
     *
     *p r i n t _ i t e m
     *
     ************************************
     *
     * Functional description
     * Print a SQL data item as described.
     *
     ************************************/
    char     *p, *string;
```

```
char      d[20];
VARY     *vary;
short      dtype;
struct tm  times;

p = *s;

dtype = var->sqltype & ~1;
if ((var->sqltype & 1) && (*var->sqlind < 0))
    /* If field was missing print <null> */
    {
    if ((dtype == SQL_TEXT) || (dtype == SQL_VARYING))
        sprintf (p, "%-*s ", var->sqllen, "<null>");
    else if (dtype == SQL_DATE)
        sprintf (p, "%-20s ", "<null>");
    else
        sprintf (p, "%*s ", var->sqllen, "<null>");
    }
else
    {
    /* Print field according to its data type */
    if (dtype == SQL_SHORT)
  sprintf (p, "%6d ", *(short*) (var->sqldata));
    else if (dtype == SQL_LONG)
  sprintf (p, "%11ld ", *(long*) (var->sqldata));
    else if (dtype == SQL_FLOAT)
  sprintf (p, "%f ", *(float*) (var->sqldata));
    else if (dtype == SQL_DOUBLE)
  sprintf (p, "%f ", *(double*) (var->sqldata));
    else if (dtype == SQL_TEXT)
  sprintf (p, "%.*s ", var->sqllen, var->sqldata);
    else if (dtype == SQL_DATE)
        {
        gds_$decode_date (var->sqldata, &times);
  sprintf (d, "%2d-%s-%4d ", times.tm_mday, alpha_months
[times.tm_mon],
       times.tm_year + 1900);
        sprintf (p, "%-20s ", d);
        }
    else if (dtype == SQL_VARYING)
  {
  vary = (VARY*) var->sqldata;
        string = vary->vary_string;
```

```
            string[vary->vary_length] = 0;
        sprintf (p, "%-*s ",
            var->sqllen, vary->vary_string);
        }
          }

    while (*p)
        ++p;

    *s = p;
    }


    static print_line (sqlda)
        SQLDA  *sqlda;
    {
    /************************************
     *
     *p r i n t _ l i n e
     *
     ************************************
     *
     * Functional description
     *Print a line of SQL variables.
     *
     ************************************/
    char    *p, line [1024];
    SQLVAR   *var, *end;

    p = line;

    for (var = sqlda->sqlvar, end = var + sqlda->sqld; var < end;
    var++)
        print_item (&p, var);

    *p++ = '\n';
    *p = 0;
    printf (line);
    }



    static process_statement (string, sqlda)
```

```
    char  *string;
    SQLDA  *sqlda;
{
/*************************************
 *
 *p r o c e s s _ s t a t e m e n t
 *
 *************************************
 *
 * Functional description
 *Prepare and execute a dynamic SQL statement.
 *
 *************************************/
long   buffer [512];
short   length, lines, alignment, type, offset;
SQLVAR  *var, *end;

EXEC SQL WHENEVER SQLERROR GO TO punt;
EXEC SQL PREPARE Q1 INTO sqlda FROM :string;

/* If the statement isn't a select, execute it and be done */

if (!sqlda->sqld)
    {
    EXEC SQL EXECUTE Q1;
    return;
    }
else if (sqlda->sqld > sqlda->sqln)
    {
    printf ("Statment not executed, can only select %d
          items\n", sqlda->sqln);
    return;
    }

/* Otherwise, open the cursor to start things up */

EXEC SQL DECLARE C CURSOR FOR Q1;
EXEC SQL OPEN C;

/* Set up SQLDA to receive data */

offset = 0;
```

```
for (var = sqlda->sqlvar, end = var + sqlda->sqld; var < end;
var++)
    {
    alignment = var->sqllen;
    type = var->sqltype & ~1;
    if (type == SQL_BLOB)
        {
        printf ("Statment not executed, cannot select blob
fields\n");
        return;
        }
    if (type == SQL_TEXT)
  alignment = 1;
    else if (type == SQL_VARYING)
  alignment = sizeof (short);
    offset = ALIGN (offset, alignment);
    var->sqldata = (char*) buffer + offset;
    offset += var->sqllen;
    offset = ALIGN (offset, sizeof (short));
    var->sqlind = (short*) ((char*) buffer + offset);
    offset += sizeof  (short);
    }

/* Fetch and print records until EOF */

for (lines = 0;; ++lines)
    {
    EXEC SQL FETCH C USING DESCRIPTOR sqlda;
    if (SQLCODE)
  break;
    if (!lines)
  printf ("\n");
    print_line (sqlda);
    }

if (lines)
    printf ("\n");

EXEC SQL CLOSE C;
EXEC SQL WHENEVER SQLERROR CONTINUE;
return;

punt:
```

```
    /* printf ("Statement failed, SQLCODE = %d\n\n", SQLCODE); */
     gds_$print_status (gds_$status);
}
```

COBOL Example

Executing a Known Statement

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MSQL2.
*
* copyright (c) 1989 by Interbase Software Corporation
*
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
* Include the SQL Communications Area */
    EXEC SQL
         INCLUDE SQLCA
    END-EXEC

01  ADDR PIC S9(9) USAGE COMP EXTERNAL.
01  FLAG1 PIC S9(4) USAGE COMP.
01  FLAG2 PIC S9(4) USAGE COMP.
01  FLAG3 PIC S9(4) USAGE COMP.
01  D-STAT PIC ZZZ9.
01  D-POP PIC ZZZ,ZZZ,ZZ9.
01  CITY PIC X(25).
01  STATE PIC XX.
01  POPULATION PIC S9(9) USAGE COMP.
01  SQLDA.
    02 SQLDAID PIC X(8).
    02 SQLDABC PIC S9(9) USAGE IS COMP.
    02 SQLN PIC S9(4) USAGE IS COMP VALUE IS 3.
    02 SQLD PIC S9(4) USAGE IS COMP.
    02 SQLVAR OCCURS 3 TIMES.
       03 SQLTYPE PIC S9(4) USAGE IS COMP.
       03 SQLLEN PIC S9(4) USAGE IS COMP.
       03 SQLDATA PIC S9(9) USAGE IS COMP.
       03 SQLIND PIC S9(9) USAGE IS COMP.
       03 SQLNAME_LENGTH PIC S9(4) USAGE IS COMP.
       03 SQLNAME PIC X(30).
01  QUERY PIC X(61).
PROCEDURE DIVISION.
MAIN-ROUTINE.
```

```
    MOVE "SELECT CITY, STATE, POPULATION FROM CITIES
         WHERE STATE = 'NY'"
        TO QUERY
    EXEC SQL
        WHENEVER SQLERROR GO TO ERROR_END
    END-EXEC

    EXEC SQL
        PREPARE Q INTO SQLDA FROM :QUERY
    END-EXEC

*       Establish a concurrence between the SQLDA pointer
*        variables and the program variables that will receive
*       the data.Force text fields to SQL_TEXT (452), because
*       COBOL won't handle VARYING TEXT gracefully.
*       Make that 453 because the field has an indicator
*       variable and that indicator variables are associated with
*       fields whose datatype is odd.

    CALL "ADDR" USING BY REFERENCE CITY GIVING SQLDATA(1).
    CALL "ADDR" USING BY REFERENCE FLAG1 GIVING SQLIND(1).
    MOVE 453 TO SQLTYPE(1).

    CALL "ADDR" USING BY REFERENCE STATE GIVING SQLDATA(2).
    CALL "ADDR" USING BY REFERENCE FLAG2 GIVING SQLIND(2).
    MOVE 453 TO SQLTYPE(2).

    CALL "ADDR" USING BY REFERENCE POPULATION GIVING SQLDATA(3)
    CALL "ADDR" USING BY REFERENCE FLAG3 GIVING SQLIND(3)

    EXEC SQL
        DECLARE C CURSOR FOR Q
    END-EXEC

    EXEC SQL
        OPEN C
    END-EXEC

    EXEC SQL
        FETCH C USING DESCRIPTOR SQLDA
    END-EXEC
```

```
    PERFORM UNTIL SQLCODE NOT = 0
        MOVE POPULATION TO D-POP
        DISPLAY CITY, STATE, D-POP
        EXEC SQL
            FETCH C USING DESCRIPTOR SQLDA
        END-EXEC
    END-PERFORM.

    EXEC SQL
        COMMIT RELEASE
    END-EXEC
    STOP RUN.

END-WORK.
ERROR_END.
    CALL "GDS_$PRINT_STATUS" USING BY REFERENCE
GDS_$STATUS_VECTOR
    STOP RUN.
```

FORTRAN Examples

Executing a Known Statement (Apollo FORTRAN)

```
C
C copyright (c) 1989 by Interbase Software Corporation
C
      integer*2 sqlda(1)
      character*8 sqldaid
      integer*4 sqldabc
      integer*2 sqln, sqld
      integer*2 sqlvar_type1, sqlvar_type2, sqlvar_type3
      integer*2 sqlvar_len1, sqlvar_len2, sqlvar_len3
      integer*4 sqlvar_data1, sqlvar_data2, sqlvar_data3
      integer*4 sqlvar_ind1, sqlvar_ind2, sqlvar_ind3
      integer*2 sqlvar_namelen1, sqlvar_namelen2,
      +       sqlvar_namelen3
      character*30 sqlvar_name1, sqlvar_name2, sqlvar_name3
      character*61 query
      character*25 city
      character*4 state
      integer*4 population
      integer*2 flag1, flag2, flag3
      common /sqldacmn/ sqldaid, sqldabc, sqln, sqld,
     *    sqlvar_type1, sqlvar_len1, sqlvar_data1, sqlvar_ind1,
     *    sqlvar_namelen1, sqlvar_name1,
     *    sqlvar_type2, sqlvar_len2, sqlvar_data2, sqlvar_ind2,
     *    sqlvar_namelen2, sqlvar_name2,
     *    sqlvar_type3, sqlvar_len3, sqlvar_data3, sqlvar_ind3,
     *    sqlvar_namelen3, sqlvar_name3
      equivalence (sqlda, sqldaid)
      EXEC SQL INCLUDE SQLCA
      data sqln /3/

      EXEC SQL WHENEVER SQLERROR GO TO 999

      query =
     * 'SELECT CITY, STATE, POPULATION FROM CITIES
              WHERE STATE = ''NY'''
      EXEC SQL PREPARE Q INTO sqlda FROM :query
```

```
C Associate SQLDA pointers with the program variables that
C will hold their data.Force all character fields to
C SQL_TEXT, because FORTRAN won't be happy with SQL_VARYING.
C Add one to the datatype to show that there is an indicator
C variable on the field.

      sqlvar_data1 = %LOC (city)
      sqlvar_ind1 = %LOC (flag1)
      sqlvar_type1 = SQL_TEXT + 1

      sqlvar_data2 = %LOC (state)
      sqlvar_ind2 = %LOC (flag2)
      sqlvar_type2 = SQL_TEXT + 1

      sqlvar_data3 = %LOC (population)
      sqlvar_ind3 = %LOC (flag3)

      EXEC SQL DECLARE C CURSOR FOR Q

      EXEC SQL OPEN C

100   continue
      EXEC SQL FETCH C USING DESCRIPTOR sqlda
      if (SQLCODE .ne. 0) go to 101
      write (*,*) city, state, population
      go to 100
101   continue
      EXEC SQL COMMIT RELEASE

      stop

999   write (*,*) 'Data base error, SQLCODE = ', SQLCODE
      CALL gds_$print_status (gds_$status)
      stop
      end
```

Executing a Known Statement (VAX FORTRAN)

```
C  Declare variables to receive data. Note: if it is not known
C  beforehand the nature of the fields to be retrieved,
C  it would be necessary to allocate buffer space for the
C  fields at run time, rather than using predeclared
```

```
C  fields.
      character*25 city
      character*4 state
      integer*4 population
      integer*2 flag1, flag2, flag3
      character*61 query

C  Setting up a SQLDA in FORTRAN is a bit painful as you can
C  see below. If you really want to use Dynamic SQL, you
C  might want to consider a language which supports
C  structured data types and pointers. If your heart is set on
C  FORTRAN, this is how you do it.

      integer*2 sqlda(1)
      character*8 sqldaid
      integer*4 sqldabc
      integer*2 sqln, sqld
      integer*2 sqlvar_type1, sqlvar_type2, sqlvar_type3
      integer*2 sqlvar_len1, sqlvar_len2, sqlvar_len3
      integer*4 sqlvar_data1, sqlvar_data2, sqlvar_data3
      integer*4 sqlvar_ind1, sqlvar_ind2, sqlvar_ind3
      integer*2 sqlvar_namelen1, sqlvar_namelen2,
     +   sqlvar_namelen3
      character*30 sqlvar_name1, sqlvar_name2, sqlvar_name3
      common /sqldacmn/ sqldaid, sqldabc, sqln, sqld,
     *    sqlvar_type1, sqlvar_len1, sqlvar_data1, sqlvar_ind1,
     *    sqlvar_namelen1, sqlvar_name1,
     *    sqlvar_type2, sqlvar_len2, sqlvar_data2, sqlvar_ind2,
     *    sqlvar_namelen2, sqlvar_name2,
     *    sqlvar_type3, sqlvar_len3, sqlvar_data3, sqlvar_ind3,
     *    sqlvar_namelen3, sqlvar_name3
      equivalence (sqlda, sqldaid)
      EXEC SQL INCLUDE SQLCA
      data sqln /3/

      EXEC SQL WHENEVER SQLERROR GO TO 999

C  Set up the query statement
      query =
     * 'SELECT CITY, STATE, POPULATION FROM CITIES
               WHERE STATE = ''NY'''

C  Prepare the query
      EXEC SQL PREPARE Q INTO sqlda FROM :query
```

```
C  Set up SQLDA to point to declared fields, forcing character
C  fields to be SQL_TEXT, since FORTRAN will frown on
C  SQL_VARYING fields.
      sqlvar_data1 = iaddr(city)
      sqlvar_ind1 = iaddr(flag1)
      sqlvar_type1 = SQL_TEXT + 1

      sqlvar_data2 = iaddr(state)
      sqlvar_ind2 = iaddr(flag2)
      sqlvar_type2 = SQL_TEXT + 1

      sqlvar_data3 = iaddr(population)
      sqlvar_ind3 = iaddr(flag3)

      EXEC SQL DECLARE C CURSOR FOR Q

      EXEC SQL OPEN C

C  Fetch and print out records
100   continue
      EXEC SQL FETCH C USING DESCRIPTOR sqlda
      if (SQLCODE .ne. 0) go to 101
      write (*,*) city, state, population
      go to 100
101   continue
      EXEC SQL COMMIT RELEASE

      stop

999   write (*,*) 'Data base error, SQLCODE = ', SQLCODE
      CALL gds_$print_status (gds_$status)
      stop
      end
```

Pascal Examples

Executing a Known Statement (Apollo Pascal)

```
program dsql (INPUT, OUTPUT);


    EXEC SQL
        INCLUDE SQLCA
var
(*  Set up the query statement *)
    query: array[1..61] of CHAR := 'SELECT CITY, STATE,
           POPULATION FROM CITIES WHERE STATE = "NY"';

(*  Declare variables to receive data. Note: you do not
 *  known beforehand the nature of the fields to be
 *  the nature of the fields to be retrieved, it would be
 *  necessary to allocate buffer space for the fields at
 *  run time, rather than using predeclared
 *  fields. *)

    city: array [1..25] of CHAR;
    state: array [1..4] of CHAR;
    population: integer32;
    flag1, flag2, flag3: integer16;

(*  Declare the SQLDA *)
    sqlda1: sqlda;

label
    999;

begin

    EXEC SQL
  WHENEVER SQLERROR GO TO 999

(*  Set length of SQLDA *)
    sqlda1.sqln := 100;

(*  Prepare the query *)
```

```
    EXEC SQL
        PREPARE Q INTO sqlda1 FROM :query;

(*  Set up SQLDA to point to declared fields and force
 *  character fields to the datatype SQL_TEXT, the InterBase
 *  varying type does not map correctly to the Apollo Pascal
 *  varying type. The notation <datatype> + 1 shows that
 *  the fields has an indicator variable.
*)

    sqlda1.sqlvars[1].sqldata := addr(city);
    sqlda1.sqlvars[1].sqlind := addr(flag1);
    sqlda1.sqlvars[1].sqltype := SQL_TEXT + 1;

    sqlda1.sqlvars[2].sqldata := addr(state);
    sqlda1.sqlvars[2].sqlind := addr(flag2);
    sqlda1.sqlvars[2].sqltype := SQL_TEXT + 1;

    sqlda1.sqlvars[3].sqldata := addr(population);
    sqlda1.sqlvars[3].sqlind := addr(flag3);

    EXEC SQL
        DECLARE C CURSOR FOR Q;

    EXEC SQL
        OPEN C;

(*  Fetch and print out records *)
    EXEC SQL
        FETCH C USING DESCRIPTOR sqlda1;
    while (SQLCODE = 0) do
        begin
        writeln (city, state, population);
        EXEC SQL
            FETCH C USING DESCRIPTOR sqlda1;
        end;

    EXEC SQL
  COMMIT RELEASE;

    return;

999: writeln ('Data base error, SQLCODE = ', SQLCODE);
```

```
    gds_$print_status (gds_$status);

end.
```

Executing a Known Statement (VAX Pascal)

```
(*
 * copyright (c)  1989 by Interbase Software Corporation
 *)
program dsql (INPUT, OUTPUT);

    EXEC SQL
        INCLUDE SQLCA
var
    sqlda1: sqlda;
    query: packed array[1..61] of CHAR :=
         'SELECT CITY, STATE, POPULATION FROM CITIES
              WHERE STATE = "NY"';
    city: [VOLATILE] varying [25] of CHAR;
    state: [VOLATILE] varying [5] of CHAR;
    population: [VOLATILE] integer;
    flag1, flag2, flag3: [VOLATILE] gds_$short;

label
    999;

begin

    EXEC SQL
  WHENEVER SQLERROR GO TO 999

    sqlda1.sqln := 100;
    EXEC SQL
        PREPARE Q INTO sqlda1 FROM :query;

(*  Associate each sqlda pointer with the program variable
    which will hold the data. Since we happen to know that
    the character fields are VARYING, their target variables
    are declared as VARYING. and the whole thing should run
    and the whole thing should run beautifully, since
    VAX/PASCAL is on language that supports the InterBase
    style varying text. Add one to the datatype declaration
    to show that there is an indicator variable associated
```

```
        with the field.
    *)

        sqlda1.sqlvars[1].sqldata := address(city);
        sqlda1.sqlvars[1].sqlind := address(flag1);
        sqlda1.sqlvars[1].sqltype := SQL_VARYING + 1;

        sqlda1.sqlvars[2].sqldata := address(state);
        sqlda1.sqlvars[2].sqlind := address(flag2);
        sqlda1.sqlvars[2].sqltype := SQL_VARYING + 1;

        sqlda1.sqlvars[3].sqldata := address(population);
        sqlda1.sqlvars[3].sqlind := address(flag3);

        EXEC SQL
            DECLARE C CURSOR FOR Q;

        EXEC SQL
            OPEN C;

        EXEC SQL
            FETCH C USING DESCRIPTOR sqlda1;
        while (SQLCODE = 0) do
            begin
            writeln (city, ' ', state, ' ', population);
            EXEC SQL
                FETCH C USING DESCRIPTOR sqlda1;
            end;

        EXEC SQL
      COMMIT RELEASE;

    999: if (SQLCODE <> 0) then
         begin
           writeln ('Data base error, SQLCODE = ', SQLCODE);
           gds_$print_status (gds_$status);
         end;

    end.
```

Executing an Unknown Statement (Apollo Pascal)

```
program sql (INPUT, OUTPUT);


DATABASE DB = COMPILETIME "atlas.gdb" RUNTIME db_name;

type
   statement_type = array [1..2000] of char;
   string_type = packed array [1..200] of char;

var
   done : boolean;
   sqlda1 : sqlda;
   statement : statement_type;
   db_name : packed array [1..128] of char;

procedure get_statement (VAR statement : statement_type);

(************************************
 *
 *   g e t _ s t a t e m e n t
 *
 ************************************
 *
 * Functional description
 *
 ************************************)
var
   done, more : boolean;
   i, j, k : integer;
   buffer : packed array [1..200] of char;

begin

write ('SQL> ');
readln (buffer);
done := FALSE;
j := 0;
while NOT done do
   begin
         i := 200;
   more := FALSE;
```

```
          while (i > 0) AND (buffer[i] = ' ') do
        i := i - 1;
          if i <> 0 then
        begin
        if buffer[i] = '-' then
            begin
            more := TRUE;
            i := i - 1;
            end;
        for k := 1 to i do
            statement [j + k] := buffer [k];
        j := j + i;
        if NOT more then
            begin
            for k := j + 1 to size (statement) do
                statement [k] := ' ';
            done := TRUE;
            end;
        end;
    if NOT done then
        begin
        write ('CON> ');
        readln (buffer);
        end;
    end;
end;

procedure print_line (VAR sqlda1: sqlda);

(*************************************
 *
 *   p r i n t _ l i n e
 *
 *************************************
 *
 * Functional description
 *
 *************************************)
var
   len, i, typ : integer;
   string_ptr : ^string_type;
   short_ptr : ^integer16;
   long_ptr : ^integer32;
```

```
  double_ptr : ^double;
  float_ptr : ^real;

begin

for i := 1 to sqlda1.sqld do begin
    short_ptr := sqlda1.sqlvars[i].sqlind;
    typ := sqlda1.sqlvars[i].sqltype - 1;
    if (short_ptr^ < 0) then
        begin
       (* If field was missing print <null> *)
        case typ of
            SQL_SHORT    : len := 8;
            SQL_LONG     : len := 12;
              SQL_FLOAT    : len := 12;
            SQL_DOUBLE   : len := 12;
            SQL_TEXT     : len := sqlda1.sqlvars[i].sqllen;
            end;

        write ('<null>');
        len := len - 6;
        if len > 0 then
            write (' ' : len);
        end
    else begin
        (* Print field according to its data type *)
        case typ of
            SQL_SHORT    : begin;
                           short_ptr :=
                               sqlda1.sqlvars[i].sqldata;
                           write (short_ptr^);
                           end;
            SQL_LONG     : begin;
                           long_ptr :=
                               sqlda1.sqlvars[i].sqldata;
                           write (long_ptr^);
                           end;
              SQL_FLOAT    : begin;
                           float_ptr :=
                               sqlda1.sqlvars[i].sqldata;
                           write (float_ptr^);
                           end;
            SQL_DOUBLE   : begin;
```

```
                                double_ptr :=
                                    sqlda1.sqlvars[i].sqldata;
                                write (double_ptr^);
                                end;
                        SQL_TEXT    : begin;
                                len := sqlda1.sqlvars[i].sqllen;
                                string_ptr :=
                                    sqlda1.sqlvars[i].sqldata;
                                write (string_ptr^ : len);
                                end;
                        end;
                end;
            if i <> sqlda1.sqld then
                write (' ');
            end;
        writeln (' ');
        end;

        procedure process_statement (VAR statement : statement_type;
                     VAR sqlda1: sqlda);

        (*************************************
         *
         *   p r o c e s s _ s t a t e m e n t
         *
         *************************************
         *
         * Functional description
         *
         *************************************)
        var
          done, first : boolean;
          i, typ : integer;
          string_ptr : ^string_type;
          short_ptr :  ^integer16;
          long_ptr : ^integer32;
          double_ptr :  ^double;
          float_ptr :  ^real;
        label
            999;

        begin
```

```
EXEC SQL WHENEVER SQLERROR GO TO 999;
EXEC SQL PREPARE Q1 INTO sqlda1 FROM :statement;

if sqlda1.sqld = 0 then
    begin
    (* If the statement isn't a select, execute it and be done *)
    EXEC SQL EXECUTE Q1;
    goto 999;
    end
else if sqlda1.sqld > sqlda1.sqln then
    begin
    write ('Statment not executed, can only select ');
    write (sqlda1.sqln);
    writeln (' items');
    goto 999;
    end;

(* Otherwise, open the cursor to start things up *)
EXEC SQL DECLARE C CURSOR FOR Q1;
EXEC SQL OPEN C;

(* Set up SQLDA to receive data *)
for i := 1 to sqlda1.sqld do begin
    typ := sqlda1.sqlvars[i].sqltype - 1;
    case typ of
        SQL_SHORT    : begin;
                       new (short_ptr);
                       sqlda1.sqlvars[i].sqldata := short_ptr;
                       end;
        SQL_LONG     : begin;
                       new (long_ptr);
                       sqlda1.sqlvars[i].sqldata := long_ptr;
                       end;
          SQL_FLOAT    : begin;
                       new (float_ptr);
                       sqlda1.sqlvars[i].sqldata := float_ptr;
                       end;
        SQL_DOUBLE   : begin;
                       new (double_ptr);
                       sqlda1.sqlvars[i].sqldata := double_ptr;
                       end;
        SQL_TEXT     : begin;
                       new (string_ptr);
```

```
                        sqlda1.sqlvars[i].sqldata := string_ptr;
                        end;
            SQL_VARYING : begin;
                        sqlda1.sqlvars[i].sqltype :=
                            SQL_TEXT + 1;
                        sqlda1.sqlvars[i].sqllen :=
                            sqlda1.sqlvars[i].sqllen - 2;
                        new (string_ptr);
                        sqlda1.sqlvars[i].sqldata := string_ptr;
                        end;
            SQL_DATE    : begin;
                        sqlda1.sqlvars[i].sqltype :=
                            SQL_TEXT + 1;
                        sqlda1.sqlvars[i].sqllen := 12;
                        new (string_ptr);
                        sqlda1.sqlvars[i].sqldata := string_ptr;
                        end;
            SQL_BLOB    : begin;
                        writeln
           ('Statment not executed, cannot select blob fields');
                        goto 999;
                        end;
            end;
        new (short_ptr);
        sqlda1.sqlvars[i].sqlind := short_ptr;
        end;

    (* Fetch and print records until EOF *)
    first := TRUE;
    done := FALSE;
    while NOT done do
        begin
        EXEC SQL FETCH C USING DESCRIPTOR sqlda1;
        if SQLCODE <> 0 then
      done := TRUE
        else
            print_line (sqlda1);
        end;

    if NOT first then
        writeln (' ');

    EXEC SQL CLOSE C;
```

```
EXEC SQL WHENEVER SQLERROR CONTINUE;

(* Now free the allocated variables *)
for i := 1 to sqlda1.sqld do begin
    typ := sqlda1.sqlvars[i].sqltype - 1;
    case typ of
        SQL_SHORT    : begin;
                       short_ptr := sqlda1.sqlvars[i].sqldata;
                       dispose (short_ptr);
                       end;
        SQL_LONG     : begin;
                       long_ptr := sqlda1.sqlvars[i].sqldata;
                       dispose (long_ptr);
                       end;
        SQL_FLOAT    : begin;
                       float_ptr := sqlda1.sqlvars[i].sqldata;
                       dispose (float_ptr);
                       end;
          SQL_DOUBLE   : begin;
                       double_ptr := sqlda1.sqlvars[i].sqldata;
                       dispose (double_ptr);
                       end;
        SQL_TEXT     : begin;
                       string_ptr := sqlda1.sqlvars[i].sqldata;
                       dispose (string_ptr);
                       end;
        end;
    short_ptr := sqlda1.sqlvars[i].sqlind;
    dispose (short_ptr);
    end;

999: if (SQLCODE <> 0) then
        begin
        writeln ('Data base error, SQLCODE = ', SQLCODE);
        gds_$print_status (gds_$status);
        end;
end;

begin  (* main *)

sqlda1.sqln := 100;

(* Get the runtime database name. This is the db we'll process
```

```
   queries against *)
write('Database to query?  ');
readln(db_name);

READY;
START_TRANSACTION;

(* Prompt for SQL statements and process them *)
done := FALSE;
while NOT done do
    begin
    get_statement (statement);

    if (((statement[1] = 'E') OR (statement[1] = 'e')) AND
        ((statement[2] = 'X') OR (statement[2] = 'x')) AND
        ((statement[3] = 'I') OR (statement[3] = 'i')) AND
        ((statement[4] = 'T') OR (statement[4] = 't'))) then
         begin
   COMMIT;
   FINISH;
   done := TRUE;
         end
    else if (((statement[1] = 'Q') OR (statement[1] = 'q')) AND
        ((statement[2] = 'U') OR (statement[2] = 'u')) AND
        ((statement[3] = 'I') OR (statement[3] = 'i')) AND
        ((statement[4] = 'T') OR (statement[4] = 't'))) then
         begin
   ROLLBACK;
   FINISH;
   done := TRUE;
         end
    else
        process_statement (statement, sqlda1);
    end;
end.
```

PL/1 Example

Executing a Known Statement

```
DSQL:PROCEDURE OPTIONS (MAIN);
/*
 * copyright (c) 1989 by Interbase Software Corporation
 */


    EXEC SQL
        INCLUDE SQLCA

    DECLARE CITY CHARACTER(25) VARYING;
    DECLARE STATE CHARACTER(4) VARYING;
    DECLARE POPULATION FIXED BINARY (31);
    DECLARE (FLAG1, FLAG2, FLAG3) FIXED BINARY (15);
    DECLARE QUERY CHARACTER(61) INITIAL
         ('SELECT CITY, STATE, POPULATION FROM CITIES
                   WHERE STATE = "NY"');

    DECLARE 1 SQLDA1,
      2 SQLAID CHAR(8),
      2 SQLDABC BIN FIXED(31),
      2 SQLN BIN FIXED(15),
      2 SQLD BIN FIXED(15),
      2 SQLVAR (3),
        3 SQLTYPE BIN FIXED(15),
        3 SQLLEN BIN FIXED(15),
        3 SQLDATA PTR,
        3 SQLIND PTR,
        3 SQLNAME CHAR(30) VAR;

    SQLDA1.SQLAID = 'SQLDA   ';
    SQLDA1.SQLN = 3;

    EXEC SQL
         WHENEVER SQLERROR GO TO ERR;

    EXEC SQL
        PREPARE Q INTO sqlda1 FROM :query;
```

```
/*
    associate the pointers in the SQLDA with the
    program variables that will receive their values.
    Since PL/I supports varying strings, ask for all
    text fields as SQL_VARYING.  Note that the
    form <datatype> + 1 shows that the field has an
    indicator variable.
*/

    sqlda1.sqlvar(1).sqldata = ADDR(city);
    sqlda1.sqlvar(1).sqlind = ADDR(flag1);
    sqlda1.sqlvar(1).sqltype = SQL_VARYING + 1;

    sqlda1.sqlvar(2).sqldata = ADDR(state);
    sqlda1.sqlvar(2).sqlind = ADDR(flag2);
    sqlda1.sqlvar(2).sqltype = SQL_VARYING + 1;

    sqlda1.sqlvar(3).sqldata = ADDR(population);
    sqlda1.sqlvar(3).sqlind = ADDR(flag3);

    EXEC SQL
        DECLARE C CURSOR FOR Q;

    EXEC SQL
        OPEN C;

    EXEC SQL
        FETCH C USING DESCRIPTOR sqlda1;
    DO WHILE (SQLCODE = 0);
        PUT LIST (CITY, STATE, POPULATION) SKIP;
        EXEC SQL
            FETCH C USING DESCRIPTOR sqlda1;
        END;

    EXEC SQL
  COMMIT RELEASE;

    RETURN;

ERR: PUT LIST ('Data base error, SQLCODE = ', SQLCODE) SKIP;
       CALL gds_$print_status (gds_$status);

    END;
```